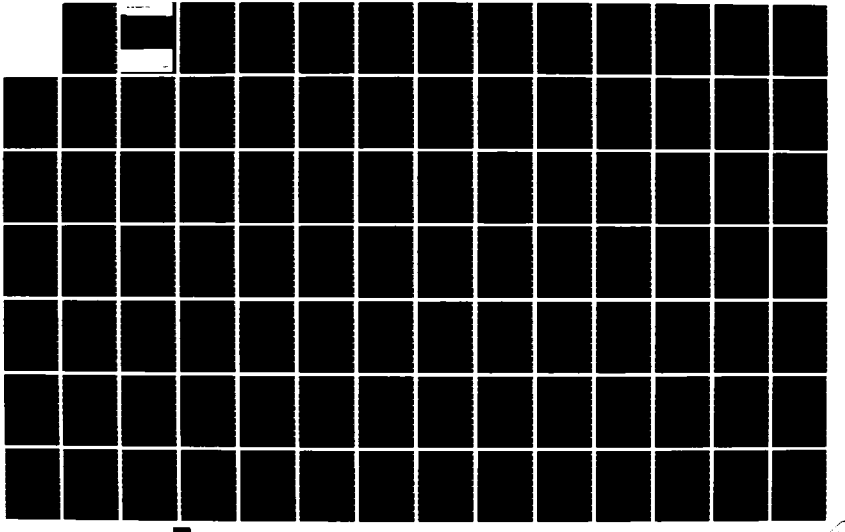
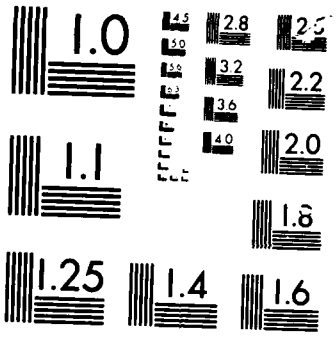
AD-A169 005 SOAR USER'S MANUAL(U) XEROX PALO ALTO RESEARCH CENTER 1/2
CA INTELLIGENT SYSTEMS LAB J E LAIRD 31 JAN 86 ISL-15
N00014-82-C-0067

UNCLASSIFIED F/G 5/10 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

Palo Alto Research Center

20

Soar User's Manual

John E. Laird

XEROX

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS AD-A169005
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISL-15	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Xerox Palo Alto Research Center 6c. ADDRESS (City, State, and ZIP Code) 3333 Coyote Hill Road Palo Alto, CA 94304	6b. OFFICE SYMBOL (If applicable) 7a. NAME OF MONITORING ORGANIZATION Personnel and Training Research Program Office of Naval Research (Code 442 PT) 7b. ADDRESS (City, State, and ZIP Code) Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable) 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-82C-0067
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO
61153N	RR042-06	RR042-06-0A	NR667-477

11. TITLE (Include Security Classification)
Soar User's Manual

12. PERSONAL AUTHOR(S)
Laird, John Edwin

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Manual	FROM 1/1/82 TO 6/15/85	January 31, 1986	106

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Cognitive Architecture, problem solving, learning, production system, problem spaces, goals.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Soar is an architecture for problem solving and learning, based on heuristic search and chunking. This manual describes Soar, version 4. This is the version of Soar currently available (January, 1986) in Common Lisp, Franz-Lisp, Interlisp and Zeta-Lisp.

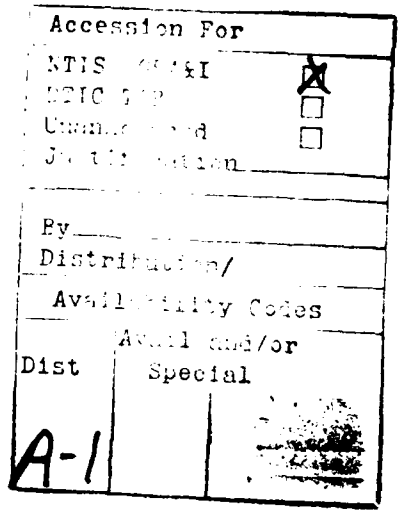
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Soar User's Manual

Version 4

John E. Laird

ISL-15 **January 1986** **[P85-00140]**



Principal researchers of the Soar Project:
John E. Laird (Xerox PARC)
Allen Newell (Carnegie-Mellon University)
Paul S. Rosenbloom (Stanford University)

The Soar software is available for non-commercial research purposes and it may be copied only for that use. Any questions concerning the use of Soar should be directed to John E. Laird at the address below. This software is made available AS IS and Xerox Corporation makes no warranty about the software, its performance, or the accuracy of this manual in describing the software. All aspects of Soar are subject to change in future releases.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539 and the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Reseach, under Contract Number N00014-82C-0067, Contract Authority Indentification Number NR 667-477. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Defense Advanced Research Projects Agency, the Office of Naval Research, or the US Government.

XEROX

Xerox Corporation
Palo Alto Research Centers
3333 Coyote Hill Road
Palo Alto, California 94304

Approved for public release:
Distribution unlimited.

Table of Contents

Preface

This manual describes *Soar*, version 4. This is the version of *Soar* currently available (January, 1986) in *Common Lisp*, *Franz-Lisp*, *Interlisp* and *Zeta-Lisp*.

Soar is an architecture for problem solving and learning, based on heuristic search and chunking. *Soar* is embedded in a production-system architecture — a modified version of *Ops5* — where all the volatile short-term information is held in *working memory* and all the fixed long-term knowledge is encoded as *productions*. Chapter 1 is an overview and introduction to the structure of the *Soar* architecture. Chapters 2 and 3 describe the nitty-gritty of working-memory representation and production representation in *Soar*. Chapter 4 describes the decision scheme that determines the selection of problem spaces, states and operators. Chapter 5 gives the details of how subgoals are automatically created and terminated. Chapter 6 describes the default processing in *Soar*, that is, the search-control knowledge that comes with *Soar*. Chapter 7 describes *chunking*, the learning mechanism in *Soar*. Chapter 8 is a short tutorial that describes how to encode goals, problem spaces, states, operators, and evaluation functions using the Eight Puzzle as an example. Chapter 9 discusses advanced programming topics. Chapter 10 describes the global variables and top-level functions of *Soar*. Chapter 11 lists all of the error and warning messages generated by *Soar* and includes some hints on correcting difficult bugs. Chapter 12 describes how to obtain and install *Soar* for different machines. Chapter 13 is a summary of benchmarking runs of *Soar* on a wide variety of computers. Chapter 14 contains an annotated bibliography of *Soar* publications. An appendix lists all of the default productions that come with *Soar*. An index is at the end of the manual. This manual does not attempt to substitute for the general scientific descriptions of *Soar* provided by the publications listed in the bibliography.

Soar is the result of joint development between John Laird, Allen Newell and Paul Rosenbloom. Credit is due to Paul Rosenbloom and Dan Scales for implementing parts of *Soar* and Ron Saul for writing the programs that convert *Soar* from *InterLisp* to the other dialects. A note of appreciation is due Lanny Forgy for creating *Ops5*, which forms the backbone of the production-system interpreter in the current implementation of *Soar*.

I would like to thank Allen Newell, Paul Rosenbloom, Jill Fain, Gregg Yost, Stephen Smoliar, Dan Scales and David Steier for comments on earlier drafts of this manual.

All suggestions, comments, and questions concerning this manual or *Soar* should be directed to
soar@h.cs.cmu.edu for computer net-mail or
John E. Laird, Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA. 94304.

1. Introduction

Soar is an architecture for general intelligence that has been applied to a variety of tasks: many of the classic artificial intelligence (AI) toy tasks such as the Tower of Hanoi, and the Blocks World; tasks that appear to involve complex, non-search reasoning, such as syllogisms, the three wise men puzzle, and sequence extrapolation; and large tasks requiring expert-level knowledge, such as the ***R1*** computer-configuration task. This chapter provides a brief overview of the ***Soar*** architecture.

In ***Soar***, every task or problem is formulated as heuristic search in a *problem space* to achieve a *goal*. A problem space consists of a set of *states* and a set of *operators* that transform one state into another. Problem solving is the process of moving from a given *initial state* in the problem space through intermediate states generated by operators until a *desired state* is reached that is recognized as attaining the goal. For each goal, there is always a single current problem space, state, and operator. The current problem space, state and operator, together with the goal, form a *context*. Goals (and their contexts) can have subgoals (and associated contexts), which form a strict goal-subgoal hierarchy. The detailed structure of these objects is described in Chapter 2.

Throughout the search, *decisions* are made to select between the available problem spaces, states, and operators. Every problem-solving episode consists of a sequence of such decisions and these decisions determine the behavior of the system. Problem solving begins with the selection of a problem space for an existing goal. This is followed by the selection of an initial state, and then an operator to apply to the state. Once the operator is selected, it is applied to create a new state. The new state can (but need not) then be selected, and the process repeats as a new operator is selected to apply to the selected state. The knowledge that implements a task — suggests feasible problem spaces, creates initial states, implements operators — is collectively called *task-implementation* knowledge. All standard weak methods can be represented as knowledge to control the selection of problem spaces, states and operators. The knowledge that controls these decisions is collectively called *search control*. Problem solving without search control is quite common, however the result is an exhaustive search of the problem space.

Figure 1-1 shows a schematic representation of the decision-making process. To bring all available task-implementation and search-control knowledge to bear on making a decision, each decision involves a monotonic *elaboration phase*. During the elaboration phase, all *directly* available knowledge relevant to the current situation is brought to bear. Knowledge that is not directly available, but can be extracted by search, can be brought to bear only in subgoals. The directly available knowledge in ***Soar*** is represented as productions. Chapter 3 describes the language for specifying productions in ***Soar***. The contexts of the goal hierarchy and their *augmentations* serve as the working memory for these productions. The information

added during the elaboration phase can take one of two forms. First, existing objects may have their descriptions augmented with new or existing objects. For example, a new state can be created that is the result of applying the current operator to the current state. Second, data structures called *preferences* can be created that assert the worth of an object for a role in a context. Each preference indicates the context in which it is relevant by specifying the goal, problem space and state.

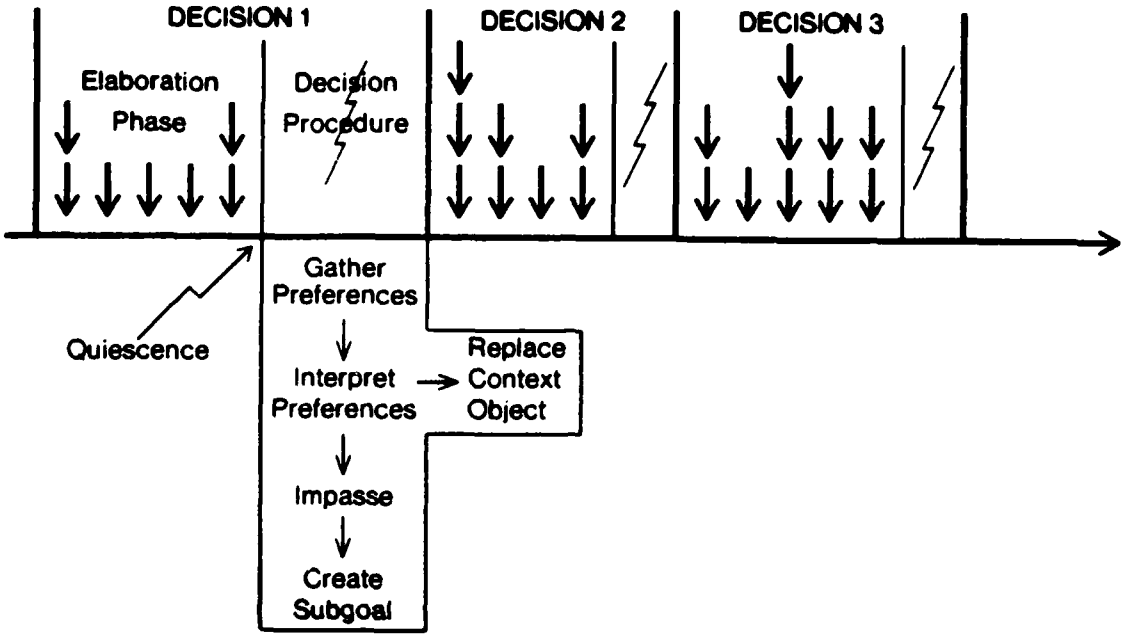


Figure 1-1: The *Soar* decision cycle.

On each cycle of the elaboration phase, all instantiations of satisfied productions fire in parallel. When the elaboration phase reaches quiescence — no more productions eligible to fire — a fixed *decision procedure* is run that integrates the preferences provided by the elaboration phase into a specific decision. The decision procedure is described in detail in Chapter 4. Starting from the oldest context, the decision procedure uses the preferences to determine if the current problem space, state and operator in each context should be changed. If sufficient knowledge is available during the search to determine a unique decision, the search proceeds unabated. However, in many cases, the directly available knowledge, encoded as productions, may be insufficient. When this occurs, because the available preferences do not determine a unique, uncontested change in a context, an *impasse* in problem solving has been reached. Four types of impasses can arise: tie (no single object was better than all of the other objects competing to change a context), conflict (two or more objects were better than each other while competing to change a context), no-change (the elaboration phase ran to quiescence without suggesting any changes to the contexts), and rejection (all competing objects were rejected, including the one currently in place).

Soar creates a subgoal (and an associated context) to resolve the impasse. Once a subgoal is created, a problem space must be selected, followed by an initial state, and then an operator. If an impasse is reached in any of these decisions, another subgoal will be created to resolve it, leading to the hierarchy of goals in ***Soar***. By treating an impasse as a subgoal, the full problem-solving power of ***Soar*** can be brought to bear to resolve the impasse, creating whatever response is appropriate for the particular instance of the impasse. These subgoals correspond to the full variety of subgoals created in standard AI systems. This ability to generate automatically all subgoals in response to impasses and to open up all aspects of problem-solving behavior to problem solving when necessary is called *universal subgoaling*. Chapter 5 gives a complete description of subgoal creation and termination in ***Soar***.

A subgoal terminates when its impasse is resolved. For example, if a tie impasse arose, it will terminate when sufficient preferences have been created so that a single object dominates the others. When a subgoal terminates, all augmentations and preferences created in that subgoal that are not connected, directly or indirectly, to a prior context are removed from working memory. Those objects that are not removed constitute the *results* of the subgoals.

Default knowledge exists in ***Soar*** to cope with the impasses, if no additional knowledge is available. For some impasses this involves rejecting a prior choice in the context; for other impasses this involves searching for knowledge to resolve the impasse. Any additional non-default knowledge about how to resolve an impasse dominates the default knowledge and controls the problem solving in the subgoal. The different default responses to impasses are described in more detail in Chapter 6.

In addition to general problem solving, ***Soar*** also supports a general learning mechanism called *chunking*. Chunking occurs as a byproduct of problem solving in goals. Whenever a goal is satisfied, a *chunk* — a production — is created that can generate the results of the goal when a similar situation recurs. The chunk's conditions are based on the working-memory elements that existed prior to the goal that were matched by the conditions of productions that fired during the processing of the goal. The chunk's actions are the working-memory elements that were created in the goal that are of potential use in the supergoal. The complete details of chunking are given in Chapter 7.

Soar is meant to be the underlying architecture for an autonomous intelligent agent. Its behavior is determined by the knowledge it contains, and ideally we should be able to describe and specify its behavior in terms of the knowledge it has for implementing and controlling its behavior. However, in this manual, the viewpoint of the user as programmer is taken. This view is more standard in programming manuals, but it is not the "true" point of view for ***Soar*** as an architecture for general intelligence.

2. Data Representation in Working Memory

The production-system aspects of ***Soar*** are derived from ***Ops5***, and as such, ***Soar*** inherits the basic representational scheme of working memory and productions provided in ***Ops5***. In this chapter, we start with a brief review of the representation of working memory in ***Ops5***, pointing out the differences in ***Soar***. Next, we describe how ***Soar*** uses this scheme to represent structures, such as goals, problem spaces, states and operators. All information on ***Ops5*** in this and the following chapters is based on the ***Ops5*** User's Manual (Forgy, 1981).

2.1. Working Memory in *Ops5*

Working memory in ***Ops5*** is a multi-set of *elements*, called *working-memory elements*. Each working-memory element consists of a *class*, followed by a set of *attribute-value* pairs. Each attribute is prefaced by a ↑. A template for a working-memory element is as follows:

(class ↑attribute1 value1 ↑attribute2 value2 . . .)

For example, a blue block that is called block3, weighs 200 grams, and is on a block called block1 could be represented as

```
(block ↑name block3 ↑color blue ↑mass 200 ↑ontop block1)
```

Each working-memory element is represented internally in ***Ops5*** as a single data structure. When a working-memory element is created (added to working memory) it is assigned a unique integer, called its *time-tag*. These time-tags are often displayed by the system in place of the working-memory element when describing sets of working-memory elements to the user. The function wm prints the working-memory element given a time-tag (see Section 10.5.6).

2.2. Working Memory in *Soar*

Working memory in ***Soar*** is a set and not a multi-set (a change from ***Ops5***). There is only one copy of a working-memory element in working memory at a time. If an action of a production tries to add an existing element to working memory, it has no effect.

In ***Soar***, there are two different types of data representations in working memory: *objects*, and *preferences*. Both of these are realized in the attribute-value representation scheme of ***Ops5***. However, the ***Ops5*** scheme has certain restrictions that force ***Soar*** to represent objects indirectly in another attribute-value scheme on top of the ***Ops5*** scheme: (1) ***Soar*** must be able to reference each individual attribute of an object without accessing the others; (2) ***Soar*** must be able to have multiple values for the same attribute of an object (a simple representation of sets); (3) multiple productions must be able to create different attributes for an object in parallel; and (4) ***Soar*** allows variables to match attributes. Each working-memory element in ***Soar*** is

an identifier-attribute-value triple (except for preferences which are described later). The class name of the working-memory element in ***Soar*** always ends in **-info** (or is just **info**). These working-memory elements are called *augmentations*. Each augmentation has three ***Ops5*** attributes: **↑identifier**, **↑attribute** and **↑value**. To avoid confusion, we will refer to the attributes of an ***Ops5*** working-memory elements as *fields*. So, in the following example, there are three fields: identifier, attribute and value. The identifier is B0003, the attributes are name and color, and the values are block3 and blue respectively.

```
(block-info ↑identifier B0003 ↑attribute name ↑value block3)
(block-info ↑identifier B0003 ↑attribute color ↑value blue)
```

To overcome the redundancy of this representation scheme, ***Soar*** provides many functions (essentially pre- and post-processors) that hide the ***Ops5*** representation by supporting a new notation called *SP* (for ***Soar*** production). For example, the above two working-memory elements would be represented as follows in SP notation:

```
(block B0003 ↑name block3 ↑color blue)
```

In SP notation, an object begins with a *class*. However, this class name is the ***Ops5*** class without **-info** (**-info** lets the user know when he is dealing with ***Ops5*** working-memory elements instead of SP objects). The SP class is translated into an ***Ops5*** class using the association list in the global variable ***sp-classes***. All classes not occurring in the list have **-info** added to them. Using this list, some ***Ops5*** classes can be represented by many SP classes. For example, **gc**, **goal**, **context**, and **goal-context** are all translated into **goal-context-info**, while **object** is translated into just **info**. Initially, there are eleven SP classes that map onto seven ***Ops5*** classes. These are pre-defined by the global variable ***sp-classes***:

```
((gc . goal-context-info) (goal . goal-context-info)
 (context . goal-context-info) (goal-context . goal-context-info)
 (problem-space . space-info) (space . space-info)
 (state . state-info) (operator . op-info) (desired . desired-info)
 (evaluation . eval-info) (object . info))
```

Following the class is the *identifier* (**B0003** above). In SP notation, the identifier must not be preceded by the attribute **↑identifier** because a working-memory element with attribute **↑identifier** is assumed to be in ***Ops5*** format. The identifier should always be a **gensymed** symbol, such as **G0023**. Following the identifier are the attribute-value pairs. Each of these pairs is an augmentation, a separate ***Ops5*** working-memory element. Thus, no single working-memory element defines all of the features of an object. Instead, the object takes its definition from the augmentations that contain its identifier.

In ***Soar***, the identifier is just an arbitrary **gensym**. If a meaningful label is desired for an object, it should be the value of the **name** attribute. The tracing facilities will use the atom in the value field of a **name** augmentation when displaying information. This makes the traces much more readable. For example:

```
(op-info ↑identifier S0012 ↑attribute name ↑value configure-backplane)
```

This would be printed by the tracing facility as **configure-backplane**.

2.3. Goal-contexts

Problem solving in ***Soar*** is controlled by *goal-contexts*. There is a strict goal hierarchy: a subgoal is only created in response to an impasse in problem solving for an active goal. Each individual context contains four *roles:* goal, problem space, state and operator. The combination of a role and a context defines a unique *slot*. The object occupying the goal role in a context is the current goal for that context; the object occupying the problem-space role is the current problem space for that context, and so on. A goal-context is represented in working memory by three augmentations of the goal identifier, one for each of the non-goal slots. These augmentations are of class **goal-context-info**. (In SP format, the class can be **goal-context**, **goal**, **context** or **gc**.) The **identifier** field contains the identifier of the goal, the **attribute** field contains the appropriate role. The **value** field contains the identifier of the object that is current for that slot. The value of a slot is **undecided** if no object has been selected for it. There is one and only one **goal-context-info** working-memory element for each slot. Only the decision procedure (to be defined later) modifies, adds, or removes these working-memory elements. Productions cannot create working-memory elements of the **goal-context-info** class that have attribute **problem-space**, **state** or **operator**.

Below is an example of the working-memory elements that define a goal-context in SP format.

```
(gc G0001 ↑problem-space G0034 ↑state G0047 ↑operator G0033)
```

This is expanded internally to three working-memory elements.

```
(goal-context-info ↑identifier G0001 ↑attribute problem-space
        ↑value G0034)
(goal-context-info ↑identifier G0001 ↑attribute state ↑value G0047)
(goal-context-info ↑identifier G0001 ↑attribute operator ↑value G0033)
```

2.4. Preferences

Preferences are a special type of data structure in ***Soar***. A preference is an assertion of the relative or absolute worth of an object for a context slot. Each preference is a single working-memory element that is of class **preference** (it is a single working-memory element in ***Ops5*** notation and also SP notation, and in both its class is **preference**). Preferences are created by productions, and they are used by the decision procedure to replace an object in a slot. The processing of preferences by the decision procedure is discussed in Chapter 4.

The fields of a preference are:

object This is the identifier of the object that the preference will affect. (In SP notation, the **↑object** preceding the identifier is optional as long as it is the first field following the class.)

role This is the name of the role that the object will play in the context: problem-space, state or operator.

value The value is a relative or absolute evaluation of the object in the object field. These evaluations are only relevant when the goal, problem-space, state and operator fields correctly match the current context. Two of the values (**acceptable** and **reject**) determine whether an object will be considered. Three of the values (**better**, **indifferent**, and **worse**) provide a comparison of an object to another object (the reference object). The remaining values equate an object to a hypothetical ideal (**best**, **indifferent**, **worst**). There is another value, **parallel**, which permits parallel execution (see Section 9.2). The exact semantics of these values are given in Section 4.

reference The identifier that is compared to the object field, only when the value field is **indifferent**, **better**, **worse**, or **parallel**.

goal, **problem-space**, **state**, and **operator**
These fields define the relevant context for the preference. A preference is only used when the current context corresponds to the context defined by these fields. If the value of one of these fields is not **nil**, it is compared to the value in the corresponding slot of a context. If all of the non-**nil** context fields of the preference match the identifiers in the corresponding slots of a context, the preference will be used in determining new values for the context.

The following preference is for an operator (x33) that has been determined to be worse than another operator (x32). Since the operator field is not specified, it becomes **nil** and the operator slot is not tested when determining the relevance of the preference.

```
(preference ↑object x33 ↑role operator ↑value worse ↑reference x32
        ↑goal g14 ↑problem-space p34 ↑state s10)
```

An object is selected for a role in a context only if there exists an acceptable-preference for that object. Thus, the acceptable-preferences for previously selected objects provide a partial history of changes to the context. Below is a short list of some of the information that is accessible via acceptable-preferences.

prior state The acceptable-preference for the current state contains the prior state in the state field.

prior operator The acceptable-preference for the current state contains the prior operator in the operator field. The prior operator is the operator used to create the current state.

result An acceptable-preference for the state role that contains the results of applying the object in the operator field to the object in the state field (which must not be **undecided**). If the operator field is **nil** or **undecided**, then it is not a result, but probably a prior state.

initial state An acceptable-preference for the state role that contains **undecided** in the state field, contains the initial states of the problem space.

3. Productions

The operation of ***Soar*** consists of a sequence of decisions with each resulting in a change to the goal-context stack. A decision consists of the elaboration phase followed by the decision procedure. During the elaboration phase, all satisfied productions fire in parallel (simulated). This continues until no more productions are satisfied. The decision procedure examines preferences and modifies the context-stack. Processing continues by returning to the elaboration phase. The details of the decision procedure are described in Section 4.

The productions in ***Soar*** can be written exactly like ***Ops5*** productions. A production consists of (1) an open parenthesis, (2) the symbol **p**, (3) the name of the production (any symbol except **nil**), (4) a set of conditions, (5) the symbol **-->**, (6) a set of actions, and (7) a close parenthesis. This production format is called P (since these productions all start with P). For example, a very simple P format production is shown below.

```
(p joe-production
   (goal-context-info ↑identifier <g> ↑attribute state ↑value <s>)
   (state-info ↑identifier <s> ↑attribute hole-shape ↑value <x>)
   (state-info ↑identifier <s> ↑attribute peg-shape ↑value <> <x>)
   -->
   (make state-info ↑identifier <s> ↑attribute fits? ↑value no))
```

Productions can also be written in SP format, which makes them much more concise. For example, the above production would be written in SP format as follows:

```
(sp joe-production
   (gc <g> ↑state <s>)
   (state <s> ↑hole-shape <x> ↑peg-shape <> <x>)
   -->
   (state <s> ↑fits? no))
```

3.1. Production Conditions

The conditions of a P format production are patterns to be matched against the elements in working memory. Each condition is a form for matching a working-memory element. In ***Soar*** a condition is a list, starting with a class name, followed by a set of attribute-value pairs. The attributes must be constants, while the class name must be a constant or a variable. The values can be one of a number of patterns. A condition is satisfied if all of its components (class and fields) can be consistently matched against a working-memory element. A production is satisfied if all of its conditions are satisfied with a consistent binding for all of the variables that appear in the conditions. A production *instantiation* is the set of working-memory elements that satisfy the production.

To simplify the matching of preferences that are relevant to a context, there is a special case for matching conditions that describe preferences. A preference is relevant to a context either if the values in its context

fields match the values of the appropriate slots or are **nil**. Therefore, a preference condition will match a preference in working memory if the values of the context fields of the working-memory element either match the values bound to the variables in the preference or are **nil** (**nil** fields are not show in working-memory elements). For example:

```
(sp x
  (gc <g> ↑problem-space <p> ↑state <s>)
  (preference <x> ↑role operator ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state <s>)
  -->
  (action ...)
```

will match

```
(gc g0001 ↑problem-space p0003 ↑state s0050)
(preference o0044 ↑role operator ↑value acceptable
        ↑problem-space p0003)
```

All of the conditions of a production should be linked, via augmentations and preferences, to one of the goal-context-infos of the production. Augmentations are one-way links, from the the identifier to the value. Preferences are one-way links, from the context fields (all must be present or **nil**) to the object. If all conditions are not linked, a warning is printed when the production is compiled.

3.1.1. Variables

A variable is a symbol that begins with a **<**, ends with a **>**, and contains an alphanumeric symbol in between. For a production to be satisfied, all occurrences of the same variable must match the same symbol or number. Two different variables can match the same symbol unless there is an explicit test that they are not equal (using **<>**).

3.1.2. Disjunctions of constants

If a set of values are contained with the symbols **<<** and **>>**, the condition will match a working-memory element with any of those symbols. Variables cannot occur within a disjunction, nor can a disjunction appear in a negated condition. There must be spaces separating both **<<** and **>>** from the symbols in between them.

```
<< red blue >>
```

would match either **red** or **blue**.

3.1.3. Predicates

There are six predicates that can precede constant or variable: **<>**, **<=>**, **<**, **<=**, **>=**, **>**. For example: **<>** **<a>**. **<>** means not equal and will match anything except the constant or variable immediately following it. **<=>** means same type and will match any symbol that is the same type (numeric or symbolic) as the constant or variable immediately following it. Similarly, **<** is less than, **<=** is less than or equal, **>=** is greater than or

equal, and > is greater than.

3.1.4. Conjunction

To signify conjunctive combinations of tests for a single field, the tests are contained within { and }. For a match to occur, all tests within the brackets must succeed.

```
{ < 50 > 20 <> <x> <y> }
```

In this example, a match would occur only if the value is less than 50, greater than 20, not equal to the value of <x> in other conditions and equal to <y> in other conditions.

3.1.5. Negated conditions

In addition to the positive tests for elements in working memory, conditions can also test for the absence of patterns. A condition preceded by "-" is called a negated condition and will be satisfied only if there does not exist a working-memory element consistent with its tests and variable bindings. A negated condition can not include a disjunction, such as << a b c >>.

3.2. Production Actions and Functions

If all of the conditions of a production are satisfied (with consistent variable bindings), the actions of the production will be performed. One significant change from ***Ops5*** is that a variable that appears only in the action of a production will automatically be bound to a new **gensymed** symbol (starting with the first letter of the variable, e.g., <s> might be bound to s1375). This symbol will be used for all occurrences of the variable in the action. This convention eliminates the need for most calls to the **bind** action.

Productions create preferences and augmentations of current objects by creating new working-memory elements. Logically, all creations occur in parallel and all satisfied productions fire in parallel, with the new working-memory elements being added during the same production cycle. The only ordering of actions is between multiple writes and accepts within a single production. Productions cannot remove or modify working-memory elements. A production should not create a working-memory element that will lead to a new instantiation of that same production because this will lead to an infinite loop. A production should only create working-memory elements that are linked — via the identifier for augmentations, and the context fields for preferences to identifiers bound — to variables in the conditions of the production. If this is not so, a warning is printed when the production is compiled.

Below are the available production actions. In the function definitions, *arg** means that any number of arguments (including zero) can be given.

Bind *arg1 arg2* Binds the value for *arg2* to *arg1*. *Arg1* must be a variable. *Arg2* can be a previously

bound variable, a constant or an action-function such as compute or accept.

```
(bind <input> (accept))
```

Call2 *F arg** — Applies function *F* to arguments *arg**. *F* and *arg** can be variables, bound to appropriate values. This is provided so that the actions of productions can control some of the top-level user functions such as watch, user-select, decide-trace, and learn.

```
(call2 watch 2)
```

Halt — Stops the execution of ***Soar***.

```
(halt)
```

Make — Adds to working memory the instantiated pattern that follows it.

```
(make state-info ↑identifier <s> ↑attribute color
                 ↑value blue)
```

Tabstop *arg1* — Binds the current tabstop being used by watch 0 to the variable *arg1*.

```
(tabstop <tab>)
(write1 (tabto <tab>) <o> |x|)
```

If <tab> is bound to 3 and <o> is bound to 4, the result is:

```
4 x
```

Write1 *arg** — Writes its arguments with blanks in between.

```
(write1 (tabto <tab>) <o> |x|)
```

If <tab> is bound to 3 and <o> is bound to 4, the result is:

```
4 x
```

Write2 *arg** — Performs the same function as **write** except that spaces are not automatically inserted between atoms.

```
(write2 (tabto <tab>) <o> |x|)
```

If <tab> is bound to 3 and <o> is bound to 4, the result is:

```
4x
```

Below are the functions that can be called within the actions.

Accept — Suspends ***Soar*** as it waits for the user to type in an atom. The result is that atom.

```
(state ↑identifier <s> ↑attribute input
       ↑value (accept))
```

Compute — Evaluates arithmetic expressions using the following five operators: + (addition), - (subtraction), * (multiplication), // (division), and \\ (modulus). Only numbers and variables bound to numbers can be used in expressions. The expressions are evaluated using standard infix notation, but there is no operator precedence. The operators are evaluated right to left, except when overridden by parentheses.

```
(state ↑identifier <s> ↑attribute sum
       ↑value (compute <x> + <y>))
(state ↑identifier <s> ↑attribute product-sum
       ↑value (compute (<v> + <w>) * (<x> + <y>)))
```

Crlf A special function that can be called within any of the write actions. It takes no arguments and forces a new line at its position in the write action.

```
(write1 <x> (crlf) <y>)
```

Tabto A special function that can be called within any of the write actions. It takes one argument that is a column number, either a number, or a variable bound to a number. It modifies the write so that it begins printing at the column given as its argument.

```
(write1 <x> (tabto <col>) <y>)
```

3.3. SP Format

The SP production format provides a set of mechanisms that allow more concise definitions, and automatic optimization of ***Soar*** productions. SP is a preprocessor, so (1) it does not fundamentally change what can and cannot be represented in ***Ops5*** productions, and (2) there is no problem with mixing together traditional productions (in ***Ops5*** format) and SP productions.

SP provides the ability to match a context in either the traditional way or by a single SP condition. A context such as

```
(goal-context-info ↑identifier <g> ↑attribute problem-space ↑value <p>)
(goal-context-info ↑identifier <g> ↑attribute state ↑value <s>)
```

can be given as is or shortened to

```
(goal-context <g> ↑problem-space <p> ↑state <s>)
```

SP provides the ability to specify the information about an object as either a set of separate conditions or as a single condition. A set of augmentations about an object such as

```
(state-info ↑identifier <s> ↑attribute color
      ↑value {<< red green >> <c>})
(state-info ↑identifier <s> ↑attribute depth ↑value > 10)
-(state-info ↑identifier <s> ↑attribute weight ↑value <> 30)
(state-info ↑identifier <s> ↑attribute leg ↑value <leg1>)
(state-info ↑identifier <s> ↑attribute leg ↑value <leg2>)
(state-info ↑identifier <s> ↑attribute name)
(state-info ↑identifier <s> ↑attribute << height width >>
      ↑value small)
```

can be given as is or shortened to

```
(state <s> ↑color {<< red green >> <c>} ↑depth > 10 -↑weight <> 30
      ↑leg <leg1> <leg2> ↑name ↑<< height width >> small)
```

Four aspects are of note. (1) It is possible to mix the two representations within the same production. (2) Whereas the final ***Ops5*** production can not have variable or disjunctive attributes, both are possible for attributes in SP, since each augmentation is a separate working-memory element where the SP attribute is actually a value in the ***Ops5*** representation. (3) Negations usually appear in front of the attribute, but can appear in front of the whole object if there is only one attribute in the object. (4) If there are multiple values for an attribute, a separate working-memory element is created for each value, giving a simple set notation.

If the first symbol after the class name is not ↑, then the condition is assumed to be in SP format. If the first symbol is a ↑, then it is assumed that it is in ***Ops5*** format. Preferences are always in ***Ops5*** format, but the **↑object** is optional if the object identifier directly follows the class; so

```
(goal-context-info ↑identifier <g> ↑attribute impasse ↑value <d>)
(preference ↑object <s> ↑role state ↑value acceptable ↑goal <g>)
```

can be shortened to

```
(goal <g> ↑impasse <d>)
(preference <s> ↑role state ↑value acceptable ↑goal <g>)
```

The same format can be used for both conditions and actions. In the actions, the placement of a **make** at the front of the object (of either format) is optional. There is a global list (in variable ***ops5-actions***) which is used to determine whether an action is a primitive action or a make.

The same format can also be used for **make**s at the top-level of ***LISP*** that initialize working memory. For example

```
(make space-info ↑identifier p ↑attribute operator ↑value op1)
(make space-info ↑identifier p ↑attribute operator ↑value op2)
```

can be given as

```
(smake space p ↑operator op1 ↑operator op2)
```

SP provides automatic condition ordering to yield more efficient productions. The following two productions show a single production in its SP form and its ordered P (***Ops5***) form.

```
(sp eight*create-new-state
  (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (problem-space <p> ↑name eight-puzzle)
  (state <s> ↑blank-binding <b1> ↑binding <b2>)
  (operator <o> ↑cell <c2>)
  (binding <b2> ↑cell <c2> ↑tile <t2>)
  (binding <b1> ↑cell <c1> ↑tile <t1>)
  -->
  (preference <s2> ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (state <s2> ↑swapped <b1> <b2> ↑binding <b3> <b4>
        ↑blank-binding <b4>)
  (binding <b3> ↑tile <t2> ↑cell <c1>)
  (binding <b4> ↑tile <t1> ↑cell <c2>))
```

```
(p eight*create-new-state
  (goal-context-info ↑identifier <g> ↑attribute problem-space
       ↑value <p>)
  (space-info ↑identifier <p> ↑attribute name ↑value eight-puzzle)
  (goal-context-info ↑identifier <g> ↑attribute state ↑value <s>)
  (goal-context-info ↑identifier <g> ↑attribute operator ↑value <o>)
  (state-info ↑identifier <s> ↑attribute blank-binding ↑value <b1>)
  (binding-info ↑identifier <b1> ↑attribute cell ↑value <c1>)
  (op-info ↑identifier <o> ↑attribute cell ↑value <c2>)
  (state-info ↑identifier <s> ↑attribute binding ↑value <b2>)
  (binding-info ↑identifier <b2> ↑attribute cell ↑value <c2>)
  (binding-info ↑identifier <b1> ↑attribute tile ↑value <t1>)
  (binding-info ↑identifier <b2> ↑attribute tile ↑value <t2>)
  -->
  (make preference ↑object <s2> ↑role state ↑value acceptable
       ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (make state-info ↑identifier <s2> ↑attribute swapped ↑value <b1>)
  (make state-info ↑identifier <s2> ↑attribute swapped ↑value <b2>)
  (make state-info ↑identifier <s2> ↑attribute binding ↑value <b3>)
  (make state-info ↑identifier <s2> ↑attribute binding ↑value <b4>)
  (make state-info ↑identifier <s2> ↑attribute blank-binding
       ↑value <b4>)
  (make binding-info ↑identifier <b3> ↑attribute tile ↑value <t2>)
  (make binding-info ↑identifier <b3> ↑attribute cell ↑value <c1>)
  (make binding-info ↑identifier <b4> ↑attribute tile ↑value <t1>)
  (make binding-info ↑identifier <b4> ↑attribute cell ↑value <c2>))
```

In addition to ordering conditions, SP also modifies a variable in the role of a goal-context-info if that variable is not used in any other conditions. The modification is to replace the variable, say **<v>** with **{ <> undecided <v> }**. This prevents the condition from matching if the role has value **undecided**.

3.4. Conjunctive Negations

The distributed representation of objects as multiple working-memory elements makes it difficult to test for the absence of an object with a set of specific features. For example, if the user wants to test if there is *not* an object in working memory that has blue toes and a blue nose, the following conditions would *not* make the right test.

```
(sp notreallycold
  context tests and other conditions
 -(state <y> ↑toes blue ↑nose blue)
  -->
  ...)
```

Assuming that **<y>** is unconstrained by the other conditions of the production, these conditions would be satisfied only if there are no objects that have blue toes and no objects that have a blue nose, while the desired behavior is to have them be satisfied only if there are no objects that have *both* blue toes and a blue nose.

One solution to this problem requires using three productions. Production p1 tests for the co-occurrence of

positive instances of the negated conditions and produces a single working-memory element that encodes the fact that both exist. Production p2 tests for the context when the original production would fire except for the negative conditions and produces a unique symbol. Finally, production p3 tests for the absence of the encoded working-memory element produced by p1 and for the presence of the one generated by p2.

```
(sp p1
   context tests and other conditions
   (state <y> ↑toes blue ↑nose blue)
   -->
   (state <y> ↑toesandnose blue))

(sp p2
   context tests and other conditions
   -->
   (state <y> ↑specialattribute value))

(sp p3
   (state <y> ↑specialattribute value)
  -(state <y> ↑toesandnose blue)
   -->
   ...)
```

A simpler and more correct solution to this problem awaits a revised implementation of the ***Ops5*** matcher used in ***Soar***.

4. Decision Procedure

The purpose of the decision procedure is to make a change to the goal-context-stack based on the preferences in working memory. The change is either the replacement of the current value of one role of an existing context, or the creation of a new context because of an impasse.

The decision procedure processes the goal-context-stack from oldest goal to newest goal (ie., from the highest supergoal to the lowest subgoal). Each role of a context is considered, starting with the problem-space and continuing through the state and operator in order. For a given slot, all preferences *relevant* to that slot are collected. A preference is relevant to a slot if all of its non-**nil** context fields (goal, problem-space, state and operator) have the same identifiers as the corresponding roles in the context and the role of the preference is the same as the role of the slot. Using these preferences, the different objects competing for a slot are compared. The decision procedure computes a final-choice for a slot according to the semantics of acceptability, rejection and the desirability ordering. The semantics of these concepts is given in Figure 4-1.

To determine the final-choice, the set of considered-choices is first determined. These are objects that are acceptable (there are relevant acceptable-preferences for them) and are not rejected (there are no relevant reject-preferences for them). Consider applying the decision procedure to the operator slot given the context and preferences in Figure 4-2. This example includes many preferences which may not arise in the normal course of problem solving, but they help exemplify the details of the decision process.

The objects with relevant acceptable-preferences are **o0001**, **o0002**, **o0004**. These acceptable-preferences differ in which fields they specify, but all of the specified fields appear in the context. Object **o0003** has an acceptable-preference, but it is not relevant to the current context since it requires that state **s0006** be selected. Even though there is a best-preference for **o0003** that is relevant, it is not a considered-choice because there is no relevant acceptable-preference. Although **o0004** is acceptable, it is also rejected, so the set of considered-choices is only **o0001**, **o0002**. From this set, the dominant, maximal choice must be determined.

Dominance is determined by the best, better, indifference, worst, and worse-preferences. An object dominates another if it is better than the other (or the other is worse) and the latter object is not also better than the former object (which is possible because conflicts are possible). A best object dominates all other non-best objects, except those that are better than it through a better-preference or worse-preference. A worst object is dominated by all other non-worst objects, except those that it is better than through a better or worse preference. The maximal-choices are those that are not dominated by any other objects. Consider our example. **o0001** is a best object, but **o0002** is better than **o0001**. **o0002** becomes the maximal-choice because it directly dominates **o0001** through a better-preference. If **o0002** were not better than **o0001**, **o0001** would be

```
Primitive predicates and functions on objects, x, y, z, ...
   current          The object that currently occupies the slot
   acceptable(x)       x is acceptable
   reject(x)           x is rejected
   (x > y)             x is better than y
   (x < y)             x is worse than y (same as y > x)
   (x ~ y)             x is indifferent to y
   (x >> y)            x dominates y = (x > y) and ¬(y > x)

Reference anchors
   indifferent(x) ⇒ ∀y [indifferent(y) ⇒ (x ~ y)]
   best(x) ⇒ ∀y [best(y) ⇒ (x ~ y)] ∧ [¬best(y) ∧ ¬(y > x) ⇒ (x > y)]
   worst(x) ⇒ ∀y [worst(y) ⇒ (x ~ y)] ∧ [¬worst(y) ∧ ¬(y < x) ⇒ (x < y)]

Basic properties
   Desirability (x > y) is transitive, but not complete or antisymmetric
   Indifference is an equivalence relationship and substitutes over >
      (x > y) and (y ~ z) implies (x > z)
   Indifference does not substitute in acceptable, reject, best, and worst.
      acceptable(x) and (x ~ y) does not imply acceptable(y),
      reject(x) and (x ~ y) does not imply reject(y), etc.

Default assumption
   All preference statements that are not explicitly mentioned and
      not implied by transitivity or substitution are not assumed to be true

Intermediate definitions
   considered-choices = {x∈objects | acceptable(x) ∧ ¬reject(x)}
   maximal(X) = {x∈X | ∀y ¬(y >> x)}
   maximal-choices = maximal(considered-choices)
   empty(X) = ¬∃x∈X
   mutually-indifferent(X) ⇔ ∀x,y∈X (x ~ y)
   random(X) = choose one element of X randomly
   select(X) = if current∈X then current else user-select(X)

Final choice
   empty(maximal-choices) ∧ ¬reject(current) ⇒ final-choice(current)
   mutually-indifferent(maximal-choices) ∧ ¬empty(maximal-choices)
         ⇒ final-choice(select(maximal-choices))

Impasse
   empty(maximal-choices) ∧ reject(current) ⇒ rejection-impasse()
   ¬mutually-indifferent(maximal-choices) ⇒ impasse(maximal-choices)
```

Figure 4-1: The semantics of preferences.

the maximal-choice. If there were neither the better-preference nor the best-preference, the maximal-choice would consist of both objects.

Once the maximal-choice for a slot is computed, the decision procedure determines whether there is a final choice or an impasse for the slot using the rules at the bottom of Figure 4-1. These rules are mutually exclusive and complete. The current object acts as a default so that a given slot will change only if the current

```
(gc g0001 ↑problem-space p0003 ↑state s0004 ↑operator o0007)

(preference o0001 ↑role operator ↑value acceptable
        ↑goal g0001 ↑problem-space p0003)
(preference o0001 ↑role operator ↑value best
        ↑goal g0001 ↑problem-space p0003)

(preference o0002 ↑role operator ↑value acceptable
        ↑goal g0001 ↑problem-space p0003 ↑state s0004 ↑operator o0007)
(preference o0002 ↑role operator ↑value better ↑reference o0001
        ↑goal g0001 ↑problem-space p0003 ↑state s0004 ↑operator o0007)

(preference o0003 ↑role operator ↑value acceptable
        ↑goal g0001 ↑problem-space p0003 ↑state s0006)
(preference o0003 ↑role operator ↑value best
        ↑goal g0001 ↑problem-space p0003 ↑state s0004)

(preference o0004 ↑role operator ↑value acceptable
        ↑problem-space p0003)
(preference o0004 ↑role operator ↑value reject
        ↑goal g0001 ↑problem-space p0003 ↑state s0004
        ↑operator undecided)
```

Figure 4-2: An example goal-context with preferences for operator selection.

object is displaced by another object. Whenever there is no maximal-choice for a slot, the current object is maintained, unless the current object is rejected, in which case a *rejection* impasse arises. If the current object is one of the maximal-choices and it is indifferent to the other maximal-choices (or it is the only maximal-choice), then the current object is maintained, since indifferent signifies that either object is appropriate. If the current object is not a maximal-choice, and the maximal-choices are mutually indifferent, the current object is displaced by one of the maximal-choices. A set of objects are mutually indifferent if all pairs in that set are indifferent. Two objects are indifferent if either there exists a binary indifferent-preference, there is a transitive set of binary indifferent-preferences containing both of them, they are both in unary indifferent-preferences, they are both in best-preferences or they are both in worst-preferences. In the current example, there is only a single maximal-choice, **o0002**, which would displace **o0007**. If all of the maximal-choices are mutually indifferent, **user-select** is tested to determine how to select between the objects. This can be either randomly, deterministically, or by the user. See Section 10.3.7 for more details.

If the current object is to be displaced by the maximal-choice, and there is not a single object (or set of indifferent objects) that dominates, then either a *tie* or *conflict* impasse arises. A *conflict* impasse arises if the objects have conflicting better and worse preferences. A *tie* impasse arises if there are no dominance relations between the maximal-choice objects. A *no-change* impasses arises if a context has been processed and none of the slots has been changed. If the current object is not displaced, or if a pre-existing impasse still exists, the decision procedure then processes the next slot, either in the current context or the next lower context if the operator slot was just processed. If a new impasse is encountered, all subgoals are terminated, a new subgoal

is created and the elaboration phase of the next decision cycle ensues. (A **tie** or **conflict** impasse is considered to be equivalent to a previous **tie** or **conflict** impasse if the objects involved in the new impasse are a subset of those in the existing impasse.)

With appropriate preferences from the elaboration phase, it is possible for a single object to result from the decision procedure, i.e., the maximal-choice set contains exactly one object, or a set of indifferent object from which a single object is chosen as describe in Section 10.3.7. When there is a single object, the change is installed, all unconsidered slots of the current context set to **undefined**, all unconsidered contexts terminated, and the elaboration phase of the next decision cycle ensues.

5. Subgoals

All subgoals in *Soar* are created automatically by the architecture when a new impasse arises in the decision procedure. There are currently four types of impasses, leading to four types of subgoals.

- A ***tie*** impasse arises if the preferences for a slot do not distinguish between competing objects.
- A ***conflict*** impasse arises if at least two objects have conflicting preferences (such as A is better than B and B is better than A) for a slot.
- A ***no-change*** impasse arises if none of the slots change value during the decision procedure.
- A ***rejection*** impasse arises if all objects with acceptable-preferences for a role also have reject-preferences.

The first two impasses, tie and conflict, are *multi-choice* impasses, because more than one object remains following the decision procedure. The last two impasses, no-change and rejection are *no-choice* impasses, because there are no objects available from which to choose. The four impasses are mutually exclusive and exhaustive.

When a new impasse is detected, *Soar* creates a **gensymed** goal symbol and an associated goal-context which includes the problem space, state and operator for the goal, as well as a set of augmentations that help define the goal. Below are the nine goal-context-info augmentations that can be created.

problem-space This contains the identifier of the current problem-space for the goal: **undecided.**

state This contains the identifier of the current state for the goal: **undecided.**

operator This contains the identifier of the current operator for the goal: **undecided.**

impasse This contains the type of impasse: **tie, conflict, no-change, rejection.**

choices This contains either **multiple**, for **tie** and **conflict** impasses, and **none**, for **no-change** and **rejection** impasses.

role For multi-choice impasses (**tie** and **conflict**), this contains the role that the choices were competing for (**problem-space**, **state**, **operator**). For **no-change** impasses, this contains the role of the last slot that is not **undefined** (**goal**, **problem-space**, **state**, **operator**). For **rejection** impasses, this contains the role of the slot just above the slot where the rejection occurred (**goal**, **problem-space**, **state**). **Rejection** is defined in this way so that both **no-change** and **rejection** impasses have the same role for a similar difficulty.

item If the impasse has multiple choices, each acceptable object for the slot, that was either tied or conflicted, is included as an individual item augmentation.

supergoal This contains the identifier of the supergoal.

superoperator This contains the identifier of the superoperator. This is necessary for the subgoals that arise from parallel operators so that each subgoal is for a different parallel superoperator (see Section 9.2).

Here is an example of a goal-context that is created for a tie between three operators:

```
(gc G0012 ↑impasse tie ↑choices multiple ↑role operator
          ↑supergoal G0003 ↑superoperator undecided
          ↑problem-space undecided ↑state undecided ↑operator undecided
          ↑item O0009 ↑item O0010 ↑item O0011)
```

A subgoal terminates when its impasse is eliminated by the addition of preferences that change the results of the decision procedure for a supergoal. For example, if there is a tie subgoal between two objects, it will automatically terminate when a new preference is added to working memory that rejects one of the choices, makes one a best choice, makes one better than another, makes one a worst choice, or makes them both indifferent. If there is a tie between three objects, the tie will be broken when one of the objects (or a set of indifferent objects) dominates the others. So the subgoal will terminate if a best-preference is created for one of the objects, if one object is made better than the other two, and so on.

When a subgoal is terminated, many of the working-memory elements that were created in the subgoal are automatically removed from working memory. All working-memory elements created in the subgoal (and those created in its subgoals) that are linked, directly or indirectly, to any supergoal, will be retained. The determination of which working-memory elements to remove is done by a mark-and-sweep garbage-collection scheme. When a subgoal terminates, all working-memory elements that were created in the subgoal (and its subgoals) are collected together. All augmentations (but not preferences) whose identifier appears in one of the working-memory elements that existed prior to the subgoal are saved. This recurs by saving those elements whose identifiers appear in a saved element until no additional elements are saved. Preferences are saved if their context objects (identifiers in the goal, problem space, state, and operator fields) are **nil** or existed before the subgoal was created. All working-memory elements that were created in the subgoal, but not saved, are removed from working memory. All saved elements are considered to have been created in the supergoal for all future garbage collections.

6. Default Search Control

This chapter describes the default knowledge in *Soar*. This is encoded in a set of 51 productions that are always loaded in with a task. These productions are listed in Appendix I. The majority of this knowledge provides default responses to the impasses that can arise during problem solving. *Soar* provides default processing for every subgoal that can arise. This chapter starts with default knowledge that is applicable in all subgoals. This is followed by the default responses to the different impasses, which includes the selection problem space, evaluation subgoals and operator subgoaling.

6.1. Common Search-Control Productions

- **default*make-all-operators-acceptable**: If the current problem space is augmented with an operator (the operator is the value of a **↑operator** attribute), make an acceptable-preference for the operator with the current problem space in the problem space field, and **nil** in all other context fields.

- **default*no-operator-retry**: If there is an acceptable-preference for the current state, create a reject-preference for the operator in the **↑operator** field using the context fields for goal, problem space and state from the acceptable-preference for the current state (assuming that the operator is not **undecided** or **nil**).

- **default*backup-if-failed-state**: If there is a reject-preference for the current state, make an acceptable-preference for the state that was used to create it.

6.2. Default Knowledge for Impasses

6.2.1. Multi-choice impasses

If a subgoal is created for a tie or conflict impasse, an acceptable-preference and a worst-preference are created for the *selection* problem space. The selection problem space is used by default for all tie and conflict impasses. See Section 6.3 for more information. As backup to the selection problem space, there are additional productions that apply if a multi-choice impasse is followed by a no-choice impasse for the goal, which would arise if the selection space was rejected. If the impasse was a tie, worst-preferences are created for the items that tied by **default*problem-space-tie**, **default*state-tie**, and **default*operator-tie**. If the impasse was a conflict, reject-preferences are created for the items that conflicted by **default*problem-space-conflict**, **default*state-conflict**, and **default*operator-conflict**.

6.2.2. No-choice impasses - goal

The impasses where **↑choices** is **none** and **↑role** is **goal** are used as a signal that no progress was possible for the next higher impasse. That is, only when there is no knowledge about how to eliminate an impasse (no acceptable problem spaces are suggested, or they are all rejected) do these impasses arise. Such an impasse leads to the rejection of the last defined object in the super-context. If there is a no-choice impasse for the top goal, **default*goal-no-choices** halts *Soar*.

6.2.3. No-choice impasses - problem space, state and operator

If no problem space is selected to handle one of these subgoals (signalled by the creation of a no-choice impasse for the goal), this implies that there is no knowledge available to resolve the no-choice impasse. The default response is to reject the lowest object in the goal-context that is not **undecided**. This has the effect of allowing another choice to replace the rejected choice so that another path can be attempted, or of further rejecting a higher-choice if the rejected object was the only candidate for its slot. This is implemented by productions **default*problem-space-no-choices**, **default*state-no-choices**, and **default*operator-no-choices**.

6.2.4. No-change impasses - operator

If a no-change subgoal is created for the operator role, there are three possible reasons: (1) the conditions of the operator were not satisfied; (2) the operator is incompletely specified (needs to be instantiated); (3) the operator is too complex to be performed by productions and must be implemented in a subgoal in its own problem space. For the first option, the appropriate response is to use the same problem space and search for a state where the operator will apply (operator subgoaling). For the others, task-specific problem spaces must be available to perform the necessary computations. Because task-specific knowledge is required for the last two cases, we assume that the first is the default action; that is, an acceptable-preference and a worst-preference are created for the super-problem-space. These will be overridden by any acceptable-preferences for other problem spaces. See Section 6.5 for more details. If operator subgoaling fails, and all problem spaces for the subgoal are rejected, **default*operator-no-choices** will then reject the operator that led to the impasse.

6.3. Selection Problem Space

Whenever a multi-choice impasse is encountered, an acceptable-preference is made for the *selection* problem space. There is also a worst-preference created for it, so that any user provided problem space will be selected in its place. Both of these are created by **select*selection-space**. The states of the selection problem space may have evaluations of the tieing objects as augmentations. An initial, empty state is created by **select*create-state**. There is one operator provided with the selection space: **evaluate-object**

6.3.1. The evaluate-object operator

Evaluate-object is meant to create evaluations for the tieing or conflicting objects so that preferences can be created by comparing the evaluations of the different objects. Production **eval*select-evaluate** creates an operator instance for each object that is an **↑item** augmentation of the goal. These operators are named **evaluate-object**. When they are created, acceptable and indifferent-preferences are also created for them, so that there will be no tie between them (however, by using the user-select function, the user can choose which evaluate-object operator to apply first). The user can also have evaluate-object operators applied in parallel by loading in production **eval*parallel-evaluate** which resides in **default.soar**, but is currently commented out. See Section 9.2 for more on parallelism.

Each evaluate-object operator is created with the following three augmentations.

- **↑state**: the current state of the selection subgoal.
- **↑name**: **evaluate-object**.
- **↑object**: the identifier of the object to be evaluated.

Once an evaluate-object operator is selected as the current operator, it is augmented with further information. This information is only necessary if the operator is going to be applied, therefore it is more efficient to generate it only if the operator is selected.

- **↑role**: the role in the context for which the object is tied or conflicted (problem-space, state, or operator).
- **↑evaluation**: the identifier of an newly created object that will hold the evaluation. This is described in more detail in Section 6.3.2.
- **↑desired**: the desired of the supergoal (the one in which the impasse arose). The desired of a goal contains the **identifier** of an object that describes the desired state of the goal.
- **↑supergoal**: the identifier of the supergoal.
- **↑superproblem-space**: the identifier of the problem space selected in the supergoal.
- **↑superstate**: the identifier of the state selected in the supergoal.

These augmentations provide easy access to information required for computing evaluations.

6.3.2. Evaluation objects

As mentioned above, a new object of class evaluation is created when an evaluate-object operator is selected. It has the following augmentations.

- **↑object**: the identifier of the tied or conflicted object to be evaluated.

- **↑state**: the current state of the multi-choice subgoal.
- **↑desired**: the desired of the supergoal (the one in which the impasse arose). The desired of a goal contains the **identifier** of an object that describes the desired state of the goal.
- **↑operator**: the identifier of the evaluate-object operator of which this evaluation is an augmentation.

The evaluation object is used to hold the evaluation computed by the operator. For two-player games (such as Tic-Tac-Toe) the evaluation can also hold the side of the player to move. See Section 6.3.7 for more information.

Currently, there is default knowledge for two types of evaluations: numeric and symbolic. They are distinguished by the augmentation that is added to the evaluation object when they are computed. Numeric evaluations, such as a number between 1 and 10, are added as augmentations of the **↑numeric-value** attribute. For example, if an evaluation is computed to be 10, it might appear in working memory as:

```
(evaluation E0004 ↑object O0044 ↑state S0034 ↑desired E3330
        ↑operator O5555 ↑numeric-value 10)
```

Symbolic evaluations, such as success, failure, win, lose, or draw are added as augmentations of the **↑symbolic-value** attribute. For example, the same evaluation as above with success would be:

```
(evaluation E0004 ↑object O0044 ↑state S0034 ↑desired E3330
        ↑operator O5555 ↑symbolic-value success)
```

6.3.3. Applying the evaluate-object operator

A specific instance of evaluate-object can, but often will not have any productions that directly implement it. The production **eval*apply-evaluate** will apply, but only to fully instantiate the operator. Therefore, an operator no-change impasse will arise; and a subgoal will be created to compute the evaluation. This is discussed in Section 6.4. Once subgoals have been used to compute evaluations, chunks that have been built from the subgoals can directly compute the evaluations. Users are free to create their own productions that directly compute evaluations.

6.3.4. Terminating the evaluate-object operator

Evaluate-object is terminated by production **eval*reject-evaluate-finished**, which detects if the current evaluate-object operator is augmented with an evaluation object that has an evaluation with either a **↑numeric-value** or **↑symbolic-value** augmentation. In either case, a reject-preference is created for the evaluate-object operator. If the evaluation does not lead to the termination of the multi-choice subgoal, the reject-preference will lead to the selection of another evaluate-object operator or the failure of the problem space.

6.3.5. Comparing numeric evaluations

Once evaluations are created for tieing objects, they can be compared and preferences can be created that break the impasse. For numeric evaluations (evaluations with a **↑numeric-value** augmentation) users can write their own productions to compare the evaluations. If the objects being evaluated are operators (almost always the case) ***Soar*** provides some help. If the object in the **↑desired** augmentation of the supergoal (which is usually the desired state) is of class **evaluation** and is augmented with **↑better higher** or **↑better lower** (depending on whether a higher or lower evaluation is better), then productions **eval*prefer-higher-evaluation** and **eval*prefer-lower-evaluation** detect the appropriate **↑better** augmentations and create preferences when one evaluation is numerically greater than another. Production **eval*equal-eval-indifferent-preference** creates indifferent-preferences for objects that have evaluations that are numerically equal, independent of a **↑better** augmentation.

6.3.6. Comparing symbolic evaluations

If an evaluation has **↑symbolic-value success**, production **eval*success-becomes-best** creates a best-preference for the object that was being evaluated. This should break the tie and allow problem solving to continue. An evaluation should be marked with **↑symbolic-value success** only if it is known to be on the path to the goal, either because the goal was reached when evaluating the object or because an intermediate state was achieved that was known from prior experience (i.e., chunks) to be on the path to the goal. We will see in Section 6.4 that ***Soar*** has productions that will propagate success up a subgoal hierarchy when it is appropriate.

If an evaluation has **↑symbolic-value failure**, production **eval*failure-becomes-worst** creates a worst-preference for the object that was being evaluated. This may or may not break the tie and allow problem solving to continue. An evaluation should be marked with **↑symbolic-value failure** only if it is known not to be on a path to the goal.

6.3.7. Evaluations for two-player games

For two-player games, there are additional productions that process symbolic values **win**, **lose**, and **draw**. These depend on the state having two augmentations: **↑side** and **↑oside**. The value of the side augmentation should be a symbol, number or identifier that represents the player that is to move next in the current state. The value of the **↑oside** (other side) augmentation should represent the player that just moved. The values of **win**, **lose**, or **draw** are in relation to the player that just moved, that is, the one that is in **↑oside**. Therefore, when an evaluation object is augmented with a symbolic value of **win**, **lose**, or **draw**, the evaluation must also be augmented with **↑side** which contains the value from **↑oside** in the state. If the state is augmented with **↑win**, **↑lose**, or **↑draw**, as described in Section 6.4.3, then production **eval*move-side-to-eval** will copy the side

correctly. Once an evaluation of **win**, **lose**, or **draw** has been created, it is translated into a preference by **eval*winning-values**, **eval*winning-values2**, **eval*losing-values**, **eval*losing-values2** and **eval*draw-values**. A **win** for the side on move or a **lose** for the side that just moved becomes a best-preference, a **lose** for the side on move or a **win** for the side that just moved becomes a worst-preference, and a **draw** becomes an indifferent-preference.

6.4. Evaluation Subgoal

If an evaluate-object operator has been selected and no productions create evaluation values for it, an operator no-change impasse will arise and a subgoal will be created. In this subgoal, the context that led to the tie will be re-established and the tieing object that is an augmentation of the evaluate-object operator will be selected. This allows the problem solving to continue so that an evaluation of the success of that object can be made. For different types of objects, different amounts of the context have to be re-established. The production **eval*select-role-problem-space** is used for tied problem spaces, and it augments the current goal with the old desired and makes an acceptable-preference for the problem space attached to the evaluate-object operator in the object augmentation. The production **eval*select-role-state** is used for tied states. It augments the goal with the desired-state description (**↑desired**), creates an acceptable-preference for the super-super-problem-space (which is in the super-problem-space augmentation of the evaluate-object operator) and creates acceptable and best-preferences for the state in the object augmentation of the evaluate-object operator. Similarly, **eval*select-role-operator** re-establishes the old desired-state, problem space and state and then creates an acceptable-preferences for the operator in the object augmentation of the evaluate-object operator. The production **eval*reject-non-slot-operator** rejects all of the other operators that compete for the operator slot. This is necessary because new operator instantiations may be created in the subgoal that will compete (and possibly receive best-preferences) for the operator slot. Following this, problem solving is expected to continue until an evaluation is produced (of course, there may be many subgoals along the way to an evaluation). Once the evaluation is produced, the evaluate-object operator is rejected as described above.

6.4.1. Default evaluations

In four cases, the evaluations can be determined based on preferences created in the subgoals and not on any features of the states or operators.

1. If an operator is being evaluated and that operator is rejected for the initial state of the evaluation subgoal, production **eval*failure-if-reject-evaling-operator** will augment the evaluation with **↑symbolic-value failure**.

2. If an operator is being evaluated and the state that is created from applying that operator to the initial state of the evaluation subgoal is rejected, production **eval*failure-if-reject-state** will augment the evaluation with **↑symbolic-value failure**.

3. If an object is being evaluated below a selection problem space, there can be a tie impasse with a second selection problem space in the search for an evaluation. If during the problem solving in the second selection problem space an evaluation of **↑symbolic-value success** is produced relative to the same desired state as the original object, **eval*pass-back-success** will assign an evaluation of **↑symbolic-value success** to that original object.

4. If an operator is being evaluated below a selection problem space for a two-player game, there can be a tie impasse with a second selection problem space in the search for an evaluation. If during the problem solving in the second selection problem space an evaluation of **↑symbolic-value win** is produced for the same side as the original operator, **eval*pass-back-win** and **eval*pass-back-win2** will augment its evaluation with **↑symbolic-value win**.

6.4.2. Computing numeric evaluations

Numeric evaluations can be computed by a single production, a set of productions, or a subgoal. All of these methods must create the right augmentation of the correct object so that the rest of the productions can use it to terminate the evaluate-object operator and create preferences for the tieing objects by comparing evaluations. The correct action is to augment the evaluation object (which is the value of the **↑evaluation** augmentation of the evaluate-object operator) with **↑numeric-value** *number*. For example, your production would contain at least the following:

```
(sp your-production-name
   (gc <g> ↑problem-space <p> ↑state { <> <ss> <s> }
           ↑superoperator <so>)
   (problem-space <p> ↑name your-task-problem-space-name)
   (operator <so> ↑name evaluate-object ↑evaluation <e>
        ↑superstate <ss>)
   conditions that match features of state <s>
    -->
   (evaluation <e> ↑numeric-value your-evaluation))
```

Numeric evaluations are useful when features of a state correspond to the distance from the state to the goal and can be mapped onto either the integer or the real numbers. The value computed for each state can then be compared to the value computed for another state and a preference can be created based on the ordering of the numeric values. Complex combinations of numbers for a numeric evaluation of a state is possible using the **compute** action. For example, your-evaluation could be the addition of two other numbers: `(compute <num1> + <num2>)`. See Section 3.2 for a further description of compute.

6.4.3. Computing symbolic evaluations

The same approach that was used in numeric evaluations can also be used in symbolic evaluations, except that the correct augmentation for the evaluation object is **↑symbolic-value** instead of **↑numeric-value**. A simpler approach is also available so that the user does not have to even deal with evaluation objects. Instead of augmenting the evaluation object, the user can augment the current state of the subgoal with one of the following five attributes: **↑success**, **↑failure**, **↑win**, **↑draw**, **↑lose**. The value of these augmentations must be the

↑desired augmentation of the goal. A default production then converts these state augmentations to the corresponding symbolic-value augmentations for the evaluation object. For example, use a production like the following:

```
(sp your-production-name
   (gc <g> ↑problem-space <p> ↑state <s> ↑desired <desired>)
   conditions that detect subgoal success.
    -->
   (state <s> ↑success <desired>))
```

The production involved in the conversion is: **eval*state-to-symbolic-evaluation**.

6.4.4. Detecting success and failure

If a state for the top goal in ***Soar*** is marked with **↑success**, **↑win**, or **↑lose**, one of the following productions will cause ***Soar*** to halt: **eval*detect-success**, **eval*detect-win**, **eval*detect-lose**. If a state for the top goal in ***Soar*** is marked with **↑failure**, it will be rejected by **eval*detect-failure**.

6.5. Operator Subgoaling

If an operator has been selected but cannot be applied to the current state, a useful strategy is to create a subgoal to find a state where the operator can be applied. This strategy is called operator subgoaling (also precondition satisfaction) and is a common AI technique dating back to ***GPS***. In ***Soar***, operator subgoaling is appropriate when an operator has been selected and a no-change impasse arises. In such a situation, acceptable and worst-preferences are created for the super-problem-space for the subgoal by **opsub*try-operator-subgoaling**. If no other problem spaces are suggested for the goal, the problem space of the supergoal will be selected, allowing a search to be performed in the same problem space as the supergoal, but with a new goal — applying the currently selected operator. The presumption is that the selected operator could not apply to the current state, so another state must be found. The default productions are adequate to implement operator subgoaling, so that no additional productions must be added by the user.

Once the super-problem-space has been selected, the goal is named **operator-subgoal** and augmented with the superoperator as its **↑desired** by **opsub*go-for-it**. This establishes a convention that when the desired augmentation of a goal is an operator, then the object of the goal is to achieve a state in which the operator can be applied. **Opsub*go-for-it** also creates an acceptable-preference for the superstate. Once the superstate is selected, a reject-preference is created for the superoperator with the initial state in the state context field, by **opsub*reject-opsub*operator**, since it is known that it will not apply to it. Other operators must be available to create a new state. For every state created following the initial state, a best-preference is created for the superoperator by **opsub*select-opsub*operator** to try out the operator that led to the subgoal. If it generates a new state without going into another subgoal, an acceptable-preference for that state is created

that will be appropriate to the supercontext by **opsub*detect-direct-opsub-success** or **opsub*detect-indirect-opsub-success**. This will terminate the subgoal. If the operator leads to another subgoal, it is rejected by **opsub*reject-double-op-sub**.

7. Chunking

Learning in ***Soar*** is based on building productions that permanently cache the processing done in a subgoal. The actions of the production are based on the working-memory elements that are the results of the subgoal. The conditions of the productions are based on the working-memory elements that were present when the subgoal was created and then used in the subgoal to create the results.

A number of factors determine whether or not a chunk is created when a subgoal is terminated. A chunk is built unless one of the following conditions is true:

1. Learning is off.

2. The chunk would have no actions. (This attempts to guarantee that a chunk is not built for a subgoal that produces no results. Such a situation can arise when a supergoal terminates without the termination of all intermediate subgoals.)

3. The name of the current problem space of the subgoal is in *chunk-free-problem-spaces*. (***Chunk-free-problem-spaces*** lets the user control which problem spaces should not be chunked. It is initially empty, so that all problem spaces will be chunked. One strategy is only to learn search-control knowledge by including all task problem spaces in ***chunk-free-problem-spaces***.)

4. None of the conditions of the chunk have a class in ***chunk-classes***. ***Chunk-classes*** is set initially to (problem-space state operator). This prevents the creation of chunks that do not test any of the objects that existed before the subgoal was created. These chunks are usually very overgeneral.

5. Learning is bottom-up and a chunk was built for a subgoal of the current subgoal (possibly not the immediate subgoal).

6. The chunk is a duplicate of a chunk that is being built at the same time. The detection of duplicate chunks is done at a syntactic level, so sometimes chunks that are semantically equivalent to previous chunks will be built.

7.1. Determining Conditions and Actions

The determination of the conditions and actions of a chunk-production depends on the creation and reference of working-memory elements in a subgoal. This information is maintained automatically by ***Soar*** for each working-memory element in every goal. When a production fires, a *trace* of the production — the working-memory elements matched by its conditions and created by its actions — is saved on the **production-trace** property of the appropriate goal. The appropriate goal is the most recently created goal (lowest in the subgoal hierarchy) that occurs in the working-memory elements matched by the production. Only productions that actually add something to working memory have their traces saved. Therefore, productions that just monitor the state (have only write statements) will not affect the learning. If a

production tries to add working-memory elements that already exist, it will not affect the learning (although see ***chunk-all-paths*** for an alternative).

Chunking is complicated by the fact that context slots and subgoal augmentations are created by the architecture and not by productions. If these structures are tested, there are no associated conditions. Therefore, ***Soar*** associates with them those working-memory elements that are responsible for their creation. Below is the list of goal-context augmentations and their associated pseudo-conditions.

- **Problem space, state, or operator roles.** The acceptable-preference for the object in the role. The other preferences are not included in the production trace.
- **Item (for tie and conflict impasses).** The acceptable-preference for the object in the item.
- **Superoperator.** The goal-context-info for the operator of the supergoal.
- **Impasse rejection.** All the reject-preferences that led to the impasse.
- **Impasse no-change.** The goal-context-info for the next slot, with **undecided** as the value. (This is not used for operator no-change, since there is no next role.)
- **Choices none.** If this is a rejection impasse, all the reject-preferences that led to the impasse. If this is a no-change impasse

Negated conditions of productions that fire in a subgoal are included in a trace as follows. When a production fires, its negated conditions are fully instantiated with the appropriate values for its variables based on the rest of the data that matched the production's positive conditions. If the identifier used to instantiate the identifier field of the condition was created before the subgoal, then the instantiated negated condition is added to the trace (as a negated condition); otherwise it is ignored.

The actions of the chunk for a subgoal are taken to be those working-memory elements created in the subgoal (or its subgoals) that are accessible from the supergoal. An augmentation is accessible if its identifier existed before the subgoal was created or is in another result. A preference is accessible if all of its **non-nil** context objects (goal, problem space, state and operator) existed before the subgoal was created or is in another result. Once the total set of results is determined, it is split into subgroups such that no two subgroups share objects that were created in the subgoal. These results are logically separate and can be generated in the future by separate productions

Once the actions of a chunk have been determined, a dependency analysis of the production traces is used to determine exactly those working-memory elements that existed prior to the creation of the subgoal that were tested in creating the actions. Not all working-memory elements tested in a subgoal become conditions in a chunk, only those responsible for the actions. Specifically, those productions that created non-acceptable-

preferences will usually not be included (unless the preferences are results of the subgoal) in the dependency analysis because they contribute only to the decision scheme. For the decision scheme, only acceptable-preferences are saved in production traces.[1]

7.2. Replacing Identifiers with Variables

The working-memory elements that are used to create the conditions and actions have the identifiers of specific objects in their identifier fields. When building productions, all object identifiers are replaced by variables. All occurrences of an identifier are replaced with the same variable. This sometimes leads to a slightly overspecific chunk (two objects that did not have to be the same in the subgoal, but just happened to be the same, *must* be the same for the chunk to apply).

7.3. Removing Extraneous Conditions

Soar removes conditions where the identifier in the value field does not occur in any other condition or action of the production. This process recurs, so that a long linked-list of conditions (connected by value and identifier attributes) will be removed if the final one in the list has a value that is unique to that condition. These conditions provide little or no constraint on the match and greatly increase the number of instantiations.

7.4. Splitting Chunks Based on Duplicate Conditions

Following the removal of unnecessary conditions, it is possible that many conditions will match exactly the same working-memory elements. This is most serious when substructures are copied from one state to another. To eliminate these duplicate conditions (which cause combinatorial processing in the matcher), the production is split into multiple productions. Two (or more) conditions are duplicates if they are exactly the same except that they differ in the **↑value** field. In addition, the identifiers in both of those fields must not be referenced by any other condition and must be referenced by actions. It is assumed that these conditions are used for copying structures and do not really test an important aspect. One of these conditions is saved along with the actions that share the identifier in its **↑value** field. All of the other duplicate conditions and the actions that share the identifiers of their **↑value** field are eliminated. More than one set of duplicates can occur for a single production, and a list is maintained of the representative condition and actions for each set of duplicates.

From these lists, productions will be created. The first production built does everything the subgoal did except for processing the duplicates. This production does not contain any of the conditions or actions that

[1] This may lead to overgeneral chunks. We are currently re-examining this design choice and may modify it in the future.

were duplicates. Additional productions are built for each set of duplicates. The conditions of these productions contain:(1) all of the conditions of the first production; (2) all actions of the first production (so it won't fire until after the first and can bind to all identifier's created in the first production); and (3) the one instance of a duplicate condition saved away. The actions of the production are only those actions that were saved with the duplicate condition. Therefore, for one subgoal, many productions may be built.

7.5. Ordering Conditions

The efficiency of the Rete matcher used in ***Soar*** is heavily dependent on the order of the conditions in the productions. Therefore, ***Soar*** orders the conditions in an attempt to make the matching process more efficient. The ordering algorithm is implemented by trying to determine, at each stage, which eligible condition, if placed next, will have the fewest number of instantiations when the production is used. The details of the ordering algorithm are given in the ***Soar*** Technical Manual.

7.6. Making Different Variables Distinct

When variables were assigned to conditions, all identical identifiers were replaced by the same variable. However, the resulting production could match the same identifier to different variables, so that the semantics of the productions are incorrect. Since variables in ***Ops5*** do not have to match distinct identifiers, ***Soar*** explicitly modifies the production so that no two variables can match the same identifier. ***Soar*** also automatically modifies any goal-context-info with attribute **↑problem-space**, **↑state**, or **↑operator** that has a variable in its value field that does not appear in any other condition (but does appear in an action). The modification is to replace the variable, say **<p>**, with **{ <> undecided <p> }**.

7.7. Refractory Inhibition of Chunks

When a production is built as a part of a chunk, it may be able to fire immediately on those working-memory elements that were used to create it. If the actions of the production include the creation of new objects, the production will immediately fire and create another object, in addition to the object that was the original result of the subgoal. To avoid this, each production that is built during chunking is refracted so that it will not fire on the working-memory elements used to create it. This does not prevent a newly learned production from firing on other working-memory elements that are present.

7.8. Over-generalization

Chunking in ***Soar*** can lead to over-generalization in three ways. First, when there is special-case knowledge that is not used in solving a subgoal. This knowledge is encoded in productions for which most but not all of the conditions were satisfied during a problem-solving episode. Those that were not satisfied either tested for

the absence of something that is available in the subgoal (using a negated condition) or for the presence of something missing in the subgoal. The chunk that is built for the subgoal may be over-general because it does not include the inverses of these conditions. During a later episode, when all of the conditions of a special-case production would be satisfied in a subgoal, the chunk learned in the first trial bypasses the subgoal. If the special-case production would lead to a different result for the goal, the chunk is over-general and produces an incorrect result.

Overly general chunks can also be learned when there are negated conditions of productions in a subgoal that test for the absence of a working-memory element that would be created in the subgoal. If the creation of that working-memory element was directly related to the existence of a working-memory element that existed before the subgoal, then the test for the absence of the working-memory element local to the subgoal should be replaced by a test for the absence of the working-memory element that existed before the subgoal. Chunking is currently unable to perform such an analysis and include tests for the absence of working-memory elements unless they are explicitly made in a production. This inability can lead to overly general chunks.

When determining the conditions of a chunk via the dependency analysis, the conditions of productions that created non-acceptable preferences are included only if they were results of the subgoal, or the results were produced based on them. They are not included if the preferences only influenced the decisions during the problem solving. The theory is that these productions influence the efficiency of the search, but do not change its validity. That is the theory, but in practice, problem spaces can be implemented that depend on productions that create non-acceptable preferences. Instead of applying all tests for success (the goal test) to each state in the problem space, it is possible to move some of the goal test to productions that reject intermediate state (or operators) that do not satisfy some of the goal constraints. This allows the final goal test to be much simpler, since any state it tests is guaranteed to satisfy some of the constraints already. In these cases, the productions created by chunking are overly general because they do not include all the conditions they should since only the final goal test is included in the chunk, and not the implicit tests made during the search that guaranteed that a valid state was always chosen.

8. Encoding a Task

This chapter describes how to represent goals, problem spaces, states, operators and search control for a task. The Eight Puzzle will serve as an example. All of the productions will be in SP format, and these productions will actually perform the task. The productions will be given in lower-case, which is appropriate for all systems except ***Interlisp***.

8.1. Problem Space Decomposition

The first step in encoding a task in ***Soar*** is to decompose it into a set of problem spaces. This is a difficult step and corresponds to structuring the task. However, only a single problem space is necessary to represent and solve the Eight Puzzle. This problem space consists of states that have different configurations of eight numbered tiles in a 3x3 frame and operators that move tiles adjacent to the blank space into the blank space. In contrast, ***R1-Soar*** has a hierarchy of up to ten different problem spaces. Such a hierarchy arises when the operators of one problem space require a second problem space for their implementation. The operators of the high-level problem spaces are not implemented directly by productions, but instead are implemented by other operators in other problem spaces. At some point the hierarchy bottoms out, and the operators are implemented directly by productions.

As of yet, there are no hard and fast rules for decomposing a problem into multiple problem spaces. It is never necessary to decompose a task into separate problem spaces because every hierarchy of problem spaces can be represented as a single problem space, with search-control knowledge that simulates the control achieved through decomposition into separate problem spaces. With decomposition, it is often possible to represent a task as a set of problem spaces with little or no search control. Problem space decomposition is possible when different aspects of the state of the task can be modified independently of other parts of the state, or when different sets of operators are selected together, independently of other operators. The sets of operators that act independently can then be grouped into separate problem spaces. These problem spaces are then selected in response to no-change impasses for a high-level operator that represents the problem solving that will occur in the subgoal.

8.2. States

As in a standard programming language, the next step in designing and implementing a task is deciding on a representation of the data being manipulated. In ***Soar***, this involves defining the representation of the states of the problem spaces. Given the available attribute-value scheme, many different representations are possible for a given task. One structural restriction is that all substructure of a state must be linked to the state, either directly (through a single augmentation), or indirectly (through a chain of augmentations). The

augmentations then form a directed lattice, where the identifier of the state is the root.

The representation of the states has a large impact on the efficiency and the generality of problem solving and learning. From our experience, efficiency and generality is maximized if the implementations of operators and search control are able to test and create only those aspects of the problem that are necessary to perform the required functions. There are two general rules for implementing this principle.

1. Every piece of information that is relevant to the problem solving should be represented explicitly, either as an object, as the augmentation of an object, or in the structure of a set of augmentations. This removes the need for complex condition predicates that can detect implicit information, such as comparing two absolute positions given in a coordinate system and detecting that they are adjacent. If a piece of information is not represented explicitly, the testing or creation of that information will involve testing or creating other information. (If only the absolute positions are explicitly represented, the absolute positions must be tested to determine adjacency.)

2. Dynamic and static information should be represented separately, minimizing the amount of information that is dynamic. Dynamic information (data that can be changed by operators) should be represented by augmentations of the state. If the static information is tied directly to the state, it must be explicitly copied from state to state. When possible, static information (data that is not changed by operators) should be represented by augmentations of dynamic information. By making this separation, the static information is unchanged by operator application, minimizing the amount of processing required to apply an operator. If the static information is tied directly to the state, it must be explicitly copied from state to state.

Let's apply these two principles to the Eight Puzzle. In this example, there is only a single problem space. When there are multiple problem spaces that share the same data structures, the application of these rules is more problematic because information that is static in one problem space may be dynamic in another.

In determining an appropriate representation, the operators of a problem space must be considered because they determine what information is necessary to solve the problem and whether the information is dynamic or static. Consider the Eight Puzzle, which consists of a 3x3 frame with eight tiles, labelled 1-8, and a blank space. The nine positions that contain the tiles are called *cells*. The operators of the problem space move a tile in a cell adjacent to the blank space into the cell with the blank. A problem is to start at some initial configuration and, through a series of tile movements, obtain some desired configuration. Figure 8-0 contains an example initial and desired state.

To derive a representation that obeys both of the representational rules, we first determine the information that is used in solving the problem and therefore must be explicitly represented. Two types of knowledge are a necessary part of problem solving: (1) operator-implementation knowledge, and (2) goal-test knowledge. Each of these test different aspects of the state. Below is a list of the information required to implement the

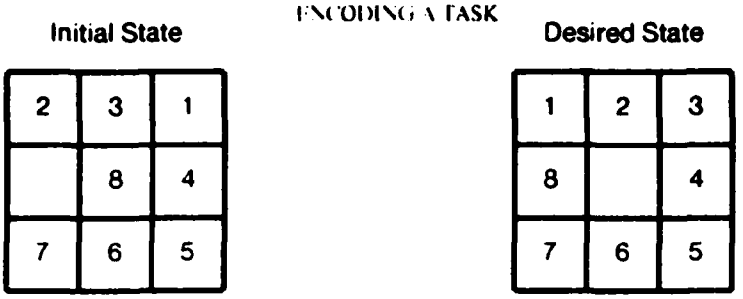


Figure 8-1: Eight Puzzle initial and desired states.

task.

1. The relative positions of the tiles and the blank. These are needed to determine if a tile is next to the blank so that the tile can be moved: operator-implementation knowledge.

2. The absolute positions of the tiles and the blank. These are needed to determine if the tiles are in the same cells as those in the desired state: goal-test knowledge.

3. The numbers on the tiles. These are needed to determine if the tiles are in the same position as those in the desired state: goal-test knowledge.

The next issue is to minimize the amount of dynamic data that must be modified when an operator applies. When an operator is applied, it changes neither the tile, nor the cell that it occupied. All it changes is the relationship between the tile and two cells on the board (the cell where it was and the cell that it now occupies). We can reify that relationship and represent it as an object. Once the relationship is an object, the operators need only manipulate the relationship and not the other objects. Let's call the relationship a **binding**, since it represents a binding of the tile to a specific cell. Therefore, a state consists of a set of nine **bindings** one for each of the tile and cell combinations. Each binding has an augmentation for a tile and a cell. Each tile is augmented with the number on it, while each cell is augmented with its absolute position. To represent the relative positions of the cells (so that the relative position of the tiles can be determined), the cells are also augmented with their adjacent cells. All the dynamic information is encoded as bindings, while all of the static information is encoded in the tile and cell objects. The operators will only manipulate bindings, and never modify the tile or cell objects. To improve the efficiency of some of the matches, the state is also augmented directly with the binding for the blank (**↑blank-binding**) and the binding of the tile that was just moved (**↑moved-tile-binding**). Below are a set of actions that create a state in this format.

```
(state <s> ↑blank-binding <bb5> ↑binding <bb0> <bb1> <bb2> <bb3>
       <bb4> <bb5> <bb6> <bb7> <bb8> ↑blank <c23>)
 (binding <bb0> ↑cell <c11> ↑tile <t2>)
  (cell <c11> ↑name 11 ↑cell <c12> ↑cell <c21>)
  (tile <t2> ↑name 2)
 (binding <bb1> ↑cell <c12> ↑tile <t1>)
  (cell <c12> ↑name 12 ↑cell <c11> ↑cell <c13> ↑cell <c22>)
  (tile <t1> ↑name 1)
 (binding <bb2> ↑cell <c13> ↑tile <t7>)
  (cell <c13> ↑name 13 ↑cell <c12> ↑cell <c23>)
  (tile <t7> ↑name 7)
 ...
```

In addition to the two rules stated earlier, there are three special cases of them that should be kept in mind when creating state representations.

1. A constant can be tested in two different ways by the productions used in solving a problem. First, a production may test that a constant is a specific value, in which case the constant would appear in the conditions of the production. In this case, the problem solving is dependent on that specific value, and any chunk built to summarize the problem solving would correctly contain that constant. In the second case, a production may test if two different objects have the same constant (an equality test). This test is performed by matching both constants by the same variable. In this case, the problem solving is independent of the specific values of the constants, being dependent only on the fact that they are equal (or not equal). A chunk would nevertheless include the specific constants because the constant is being functionally overloaded, with its specific value, and its equality relation to other constants. The solution to this problem is to have indirect pointers to constants when they will be used in equality tests. In our example, the tile numbers were not contained in the binding augmentations of the state but were represented indirectly in the tile objects. The tile-object identifiers can then be compared for equality, without referencing the exact values of the tile names. One useful convention is that constants should appear as values only in **↑name** augmentations. All other augmentations should be the identifier of another object that has a further description.

2. All functionally independent uses of a concept should be represented as separate objects. Do not overload an attribute or value with many different uses. Each use should be represented separately. For example, if the state contains the description of an algebra problem, it might have the concept left used in two different contexts, to represent expressions on the left side of equals sign and to represent terms on the left side of another operator, such as plus. These two lefts are functionally independent. However, if both of these are tested in a problem solving episode, the resulting chunks will contain tests making them dependent. That is, any tests concerning the sides of the equation will be dependent on tests of sides of the operator. This arises because chunking assumes that if the same identifier is used in multiple places (in this case, the identifier of the object named left), then a chunk must test that it is the same, even though in this example it did not have to be the same.

3. If a disjunction is used in a condition of a production, say for the names of two problem spaces (such as **<< problem-space-one problem-space-two >>**), a chunk that included a firing of that production would include a test for only one of the two names, not both. This would make the chunk less general than necessary. To solve this problem, reify the disjunction and create another augmentation for both problem spaces and then test for that augmentation. This is exactly the

reason that there is a **↑choices** augmentation for goals. Many productions used to test for **↑impasse << tie conflict >>** or **↑impasse << no-change rejection >>** and the chunks built for the subgoals would be over-specific. By adding the **↑choices** augmentation, a single augmentation can be tested that embodies the disjunction; and the disjunction is then included in the chunks.

8.3. Operator Creation

Once a representation for the states has been designed, the problem-space operators should be defined. For a given problem, many different sets of operators may be possible for essentially the same problem space. For the Eight Puzzle, there could be twenty-four operators, one for each possible movement from each cell to an adjacent cell. In such an implementation, all operators could be made acceptable for each state and then all of those that cannot apply because the blank is not in the appropriate place would be rejected. A convention in ***Soar*** is that if a problem space is augmented with an operator (such as **(problem-space p0003 ↑operator O0002)**), an acceptable-preference for that operator will automatically be made so that the operator will be considered for every state in the problem space (by production **default*make-all-operators-acceptable**). Alternatively, only those operators that are applicable to a state could be made acceptable, which we will describe in our example below. Another implementation could have four operators, one for each direction that tiles can be moved into the blank, **up**, **down**, **left**, and **right**. Those operators that do not apply to a state (because no tile exists that can be moved in that direction) could be rejected.

In our implementation of the Eight Puzzle, there is a single general operator, which moves a tile adjacent to the blank into the blank. For a given state, instantiations of this operator are created for each of the adjacent tiles. To create the operator instantiations requires a single production, shown below. Each operator has three fields: **↑name** contains the name of the operator, which is always move-tile; **↑blank-cell** for the cell that contains the blank; and **↑tile-cell** for the cell that contains the tile that will be moved into the cell with the blank. At the same time that an operator is created, an acceptable-preference is created, so that the operator can be selected to be the current operator for the context containing the **eight-puzzle** problem space and the state with which the operator was instantiated. Since operators are created only if they can apply, no additional production is required to reject inapplicable operators.

```
(sp eight*acceptable
  (gc <g> ↑problem-space <p> ↑state <s>)
  (problem-space <p> ↑name eight-puzzle)
  (state <s> ↑blank-binding <blank>)
  (binding <blank> ↑cell <c1>)
  (cell <c1> ↑cell <c2>)
 -(preference ↑role operator ↑value acceptable
       ↑goal <g> ↑problem-space <p> ↑state <s>)
  -->
  (operator <o> ↑name move-tile ↑tile-cell <c2> ↑blank-cell <c1>)
  (preference <o> ↑role operator ↑value acceptable
       ↑goal <g> ↑problem-space <p> ↑state <s>))
```

8.4. Operator Application

An operator of a problem space is applied when it is selected by the decision procedure, i.e., when its identifier replaces the existing symbol in the role of an operator. That is, whatever happens while a given identifier occupies an operator role comprises the attempt to apply that operator. Selecting an operator and installing its identifier in the operator role produces a context in which productions associated with the operator can execute (they contain a condition that tests that the operator is selected). Operator productions are just elaboration productions, used for operator application rather than for search control.

When a nonmonotonic operator (an operator that modifies the current state) is successfully applied, it must create a preference for the new state it creates. That preference includes the current goal, problem space, state and operator. Based on this preference, the new state can be selected; and the operator will not be re-applied to the state (**default*no-operator-retry** will reject the operator). If the operator is monotonic (only adds information to the state) or fails to apply, it should create a new preference for the current state, which then leads to the operator's rejection (by **default*no-operator-retry**).

To apply an instantiated operator in the Eight Puzzle requires the two productions shown below. When the identifier of a move-tile operator is selected as an operator in the **eight-puzzle** problem space, production **eight*create-new-state** will apply and create a new state with the moved tile and the blank in their new positions. It detects that there is an operator in the operator role and matches the binding (**<b1>**) for the blank tile (**<t1>**) and its cell (**<c1>**). It also matches the cell that is connected to **<c1>** via the operator (**<c2>**) and matches the tile in that cell (**<t2>**). The actions of the production are to create a new state symbol (**<s2>**), a preference for that state (with the current context in its context fields), and then swap the bindings of cell **<c1>** and **<c2>**. It marks in the state the bindings that were swapped (**↑swapped**) and the bindings that were just created, distinguishing the old and new positions of the moved tile (**↑blank-binding**, **↑moved-tile-binding**). These latter augmentations will be used by search control.

```
(sp eight*create-new-state
  (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (problem-space <p> ↑name eight-puzzle)
  (state <s> ↑binding <b1> ↑binding <b2> ↑blank-binding <b1>)
  (binding <b1> ↑tile <t1> ↑cell <c1>)
  (binding <b2> ↑tile <t2> ↑cell <c2>)
  (operator <o> ↑name move-tile ↑blank-cell <c1> ↑tile-cell <c2>)
  -->
  (preference <s2> ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (state <s2> ↑swapped <b1> ↑swapped <b2> ↑blank-binding <b3>
        ↑moved-tile-binding <b4> ↑binding <b3> ↑binding <b4>)
  (binding <b3> ↑tile <t2> ↑cell <c1>)
  (binding <b4> ↑tile <t1> ↑cell <c2>))
```

A second production, **eight*copy-unchanged**, copies over all of the bindings that did not have to be swapped.

It applies after the previous production by testing for the creation of the preference for the new state (created by **eight*create-new-state**). The test of the preference must include tests that the state and operator are not equal to **nil**, because even though **<s>** and **<o>** were previously bound in the first conditions, the preference will match if its context fields match exactly or match **nil** (so that it is easy to match those preferences that are relevant to a context).

```
(sp eight*copy-unchanged
  (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (problem-space <p> ↑name eight-puzzle)
  (preference <n> ↑role state ↑value acceptable
        ↑problem-space <p> ↑state { <> nil <s>}
        ↑operator { <> nil <o>})
  (state <s> ↑binding <b>)
  (operator <o> ↑name move-tile)
  (state <n> -↑swapped <b>)
  -->
  (state <n> ↑binding <b>))
```

This production and the previous one are typical of the types of productions used to implement simple operators in ***Soar***. One production makes the changes and creates a new state, while another (or possibly others) copies those aspects of the state unaffected by the operator. This shows how to implement an operator that changes or adds new augmentations to a state. If an operator is to delete some aspect of a state, the productions that implement it should create a new state and copy only those augmentations that are to be retained.

8.5. Goal Detection

All subgoals are terminated by the architecture, which detects the resolution of an impasse through the creation of new preferences. So, in one sense, goal detection is done automatically. However, for many subgoals (and usually the top-level goal), the decision to create a preference that resolves the impasse becomes equivalent to a goal test. In addition, when an evaluation subgoal is used, it is useful to be able to signify that a state created in the subgoal will achieve a higher-level goal. Therefore, there is default knowledge in ***Soar*** that detects when a state is augmented with success or failure with respect to a given desired state. These rules create the appropriate preferences if it is a subgoal, or terminates problem solving if it is in the top-level goal (see Section 6.4).

In detecting that a state achieves a goal, the actual test can be represented either explicitly or implicitly. Sometimes the desired states are represented explicitly as an augmentation of the goal. This augmentation would usually be created after the problem space has been selected. Alternatively, the desired states may not be explicitly represented; and instead there may be a production, a set of productions, or an operator that recognize when a given state satisfies the goal without comparing it to an explicit description. There can be any level of explicit or implicit representation in between where parts of the desired state are explicitly

represented, and parts of the goal test are embedded in productions. However, the satisfaction of a goal should be detected by a test of a state (including its augmentations) and the information tied to the goal. If other information is tested (such as aspects of the problem space or the operator), then that information belongs either in the goal or in the state. Whenever the goal is augmented with additional information to be used in the goal test, it should be encoded as an object that is the value of the **↑desired** augmentation of the goal.

Although ***Soar*** allows the detection of desired states through recognition by a production (without comparison to an explicitly represented desired state), it is not the recommended practice because it leads to the learning of overly specific chunks. The production that tests for the desired state must include conditions that test for the actual values of the constants in the state. In the Eight Puzzle this would mean testing that a specific cell had a specific tile. Any chunk built to summarize the subgoal in which the test applied would be specific to the exact desired state. Instead, a comparison can be done between an explicitly represented desired state and the current state. In this case, only the equality of the identifiers that are augmented with the constants need be tested, and not the constants themselves.[2] The resulting chunk is sensitive to the relative values of the desired state and the states in the problem space and not the exact values of the constants in the state.

For the Eight Puzzle, the desired state is explicitly represented in working memory as a state. The desired state (**<d>**) is in **↑desired** augmentations of the goal. The following production detects that the desired state has been achieved.

[2]This assumes that it is possible to coordinate the states and the desired state in the problem space so that they share the same identifiers for the constants. This is not always possible.

```
(sp eight*detect-goal
  (gc <g> ↑problem-space <p> ↑state <s> ↑desired <d>)
  (space <p> ↑name eight-puzzle)
  (state <s> ↑binding <x11> <x12> <x13> <x21> <x22> <x23>
        <x31> <x32> <x33>)
  (binding <x11> ↑cell <c11> ↑tile <o11>)
  (binding <x12> ↑cell <c12> ↑tile <o12>)
  (binding <x13> ↑cell <c13> ↑tile <o13>)
  (binding <x21> ↑cell <c21> ↑tile <o21>)
  (binding <x22> ↑cell <c22> ↑tile <o22>)
  (binding <x23> ↑cell <c23> ↑tile <o23>)
  (binding <x31> ↑cell <c31> ↑tile <o31>)
  (binding <x32> ↑cell <c32> ↑tile <o32>)
  (binding <x33> ↑cell <c33> ↑tile <o33>)
  (cell <c11> ↑name 11) (cell <c12> ↑name 12)
  (cell <c13> ↑name 13) (cell <c21> ↑name 21)
  (cell <c22> ↑name 22) (cell <c23> ↑name 23)
  (cell <c31> ↑name 31) (cell <c32> ↑name 32)
  (cell <c33> ↑name 33)
  (desired <d> ↑binding <d11> <d12> <d13> <d21> <d22> <d23>
        <d31> <d32> <d33>)
  (binding <d11> ↑cell <c11> ↑tile <o11>)
  (binding <d12> ↑cell <c12> ↑tile <o12>)
  (binding <d13> ↑cell <c13> ↑tile <o13>)
  (binding <d21> ↑cell <c21> ↑tile <o21>)
  (binding <d22> ↑cell <c22> ↑tile <o22>)
  (binding <d23> ↑cell <c23> ↑tile <o23>)
  (binding <d31> ↑cell <c31> ↑tile <o31>)
  (binding <d32> ↑cell <c32> ↑tile <o32>)
  (binding <d33> ↑cell <c33> ↑tile <o33>)
  -->
  (state <s> ↑success <d>))
```

The action is to augment the state with **↑success** and the value of **↑desired**. By including the desired, this guarantees that only those goals that share the same desired state will be terminated. Default productions handle **↑success**, so that if a top-goal is detected in a subgoal (and labeled with **↑success**), evaluations and selection subgoals are handled correctly. See Section 6.4 for more information on evaluations.

In this example, the test was performed with a single, very large production. Other options are possible: (1) test each of the bindings of a state independently in parallel, and then combine the results of those tests; or (2) test the initial state and then incrementally update the comparison based on the changes made to the state.

For many problems, the generality of chunks learned by ***Soar*** is maximized if the goal test is done incrementally. An incremental goal test involves keeping track of the differences between a state and the desired state. When a new state is created, its differences are computed based on the differences in the state it was created from and any changes to the prior state that were necessary to create the new state. When there are no differences between a state and the desired state, the goal is achieved. This improves the generality of

the conditions of a chunk built for the goal because the detection of goal achievement is based only on the parts of a state that changed, and not on the complete state. When non-incremental goal tests are used, the complete state must be tested, not just the aspects that changed. Not all goals can be tested incrementally, although any goal that has a conjunction of conditions can be. In the Eight Puzzle, the position of each tile in its desired cell can be detected independently and an incremental goal test can be used. When the initial state is selected, it is augmented with a difference that is the number of tiles that are out of place. Whenever a new state is created, its difference would be computed modifying the difference of its prior state to reflect the changes in the new state (a tile is moved into or out of its desired cell).

8.6. Initialization

In addition to defining the operator selection, operator application and goal detection rules, working memory must be initialized to an appropriate goal, problem space and initial state, so that problem solving can begin. Following a call to init-soar, working memory is empty. When ***Soar*** starts with an empty working memory, a context is created that has all of the slots set to **undecided**. This context does not have a supergoal.

One way to get a task started (as in **eight*start** below), is to use a production that detects a goal without a supergoal, and creates a preference for a new problem space, in this case, one named eight-puzzle. Since the variable **<p>** only appears in the action, it will be bound to a newly generated symbol, starting with the first letter of the variable (something like P0034). The second occurrence of **<p>** (in the preference) will use this same symbol. The goal is augmented with a name that can be tested by later productions.

```
(sp eight*start
    (gc <g> ↑problem-space undecided -↑supergoal)
    -->
    (gc <g> ↑impasse none ↑name solve-eight-puzzle)
    (problem-space <p> ↑name eight-puzzle)
    (preference <p> ↑role problem-space ↑value acceptable
                ↑goal <g>))
```

The preference created to select a problem space is only sensitive to the current goal.

Another type of initialization is available using the **init-context** function, which allows the user to set the values of the top context (see Section 10.2.3).

Production **eight*initial-desired-states** creates the initial and desired state as well as a preference for the initial state. The acceptable-preference for the initial state (**<s>**) has **undecided** in the state field so that this state will be selected only at the beginning of problem solving. If the state field were unspecified (or **nil**), the acceptable-preference would make the state a candidate at all times during problem solving in goal **<g>** and problem space **<p>**, since a preference is used whenever all of its non-**nil** context fields match the roles of a context.

```
(sp eight*initial-desired-states
  (gc <g> ↑problem-space <p> ↑state undecided
        ↑name solve-eight-puzzle)
  (problem-space <p> ↑name eight-puzzle)
  -->
  (gc <g> ↑desired <d>)
  (preference <s> ↑role state ↑value acceptable
          ↑goal <g> ↑problem-space <p> ↑state undecided)
  (state <s> ↑binding <bb0> <bb1> <bb2> <bb3>
        <bb4> <bb5> <bb6> <bb7> <bb8> ↑blank-binding <bb5>)
  (binding <bb0> ↑cell <c11> ↑tile <t2>)
  (binding <bb1> ↑cell <c12> ↑tile <t1>)
  (binding <bb2> ↑cell <c13> ↑tile <t7>)
  (binding <bb3> ↑cell <c21> ↑tile <t8>)
  (binding <bb4> ↑cell <c22> ↑tile <t6>)
  (binding <bb5> ↑cell <c23> ↑tile <t0>)
  (binding <bb6> ↑cell <c31> ↑tile <t3>)
  (binding <bb7> ↑cell <c32> ↑tile <t4>)
  (binding <bb8> ↑cell <c33> ↑tile <t5>)
  (desired <d> ↑binding <d0> <d1> <d2> <d3> <d4>
        <d5> <d6> <d7> <d8>)
  (evaluation <d> ↑better higher)
  (binding <d1> ↑cell <c11> ↑tile <t1>)
  (binding <d2> ↑cell <c12> ↑tile <t8>)
  (binding <d3> ↑cell <c13> ↑tile <t7>)
  (binding <d8> ↑cell <c21> ↑tile <t2>)
  (binding <d0> ↑cell <c22> ↑tile <t0>)
  (binding <d4> ↑cell <c23> ↑tile <t6>)
  (binding <d7> ↑cell <c31> ↑tile <t3>)
  (binding <d6> ↑cell <c32> ↑tile <t4>)
  (binding <d5> ↑cell <c33> ↑tile <t5>)
  (cell <c11> ↑name 11 ↑cell <c12> ↑cell <c21>)
  (cell <c12> ↑name 12 ↑cell <c11> ↑cell <c13> ↑cell <c22>)
  (cell <c13> ↑name 13 ↑cell <c12> ↑cell <c23>)
  (cell <c21> ↑name 21 ↑cell <c11> ↑cell <c31> ↑cell <c22>)
  (cell <c22> ↑name 22 ↑cell <c21> ↑cell <c12> ↑cell <c23> ↑cell <c32>)
  (cell <c23> ↑name 23 ↑cell <c22> ↑cell <c33> ↑cell <c13>)
  (cell <c31> ↑name 31 ↑cell <c32> ↑cell <c21>)
  (cell <c32> ↑name 32 ↑cell <c31> ↑cell <c22> ↑cell <c33>)
  (cell <c33> ↑name 33 ↑cell <c32> ↑cell <c23>)
  (tile <t0> ↑name 0)   (tile <t1> ↑name 1)   (tile <t2> ↑name 2)
  (tile <t3> ↑name 3)   (tile <t4> ↑name 4)   (tile <t5> ↑name 5)
  (tile <t6> ↑name 6)   (tile <t7> ↑name 7)   (tile <t8> ↑name 8))
```

The desired state is augmented with **↑better higher**, so that evaluations with higher values will be translated into better-preferences by **eval*prefer-higher-evaluation**. Notice that the bindings of the desired state share the same cell and tile structure as the initial state. This allows the goal test to check only the equality of these augmentations and not the equality of the names of the cells and the tiles. This improves the generality of chunking, but it is not always possible, especially when the desired and initial states are created at different times.

8.7. Monitoring States

Monitoring of states makes traces much easier to read and does not impact chunking when done with no changes to working memory. However it may require productions that are costly to match because the complete structure of the state must be matched. Another option is to use the function **trace-attributes** which enables automatic tracing (see below and Section 10.4.1). Here is a monitor production for the Eight Puzzle that will trace a state after it is generated but before it is selected. Tabstop binds its argument (**<tab>**) to the current tabstop. By using tabto with the current tabstop in a write statement, the monitoring will line up with the trace. Write2 is used in the first write command because it does not insert blanks between the atoms it prints.

```
(sp eight*monitor
  (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (problem-space <p> ↑name eight-puzzle)
  (preference <n> ↑role state ↑value acceptable
        ↑problem-space <p> ↑state <s> ↑operator { <> nil <o> })
  (operator <o> ↑cell <name>)
  (state <n> ↑binding <x11> <x12> <x13> <x21> <x22> <x23>
        <x31> <x32> <x33>)
  (binding <x11> ↑cell <c11> ↑tile <o11>)
  (cell <c11> ↑name 11) (tile <o11> ↑name <v11>)
  (binding <x12> ↑cell <c12> ↑tile <o12>)
  (cell <c12> ↑name 12) (tile <o12> ↑name <v12>)
  (binding <x13> ↑cell <c13> ↑tile <o13>)
  (cell <c13> ↑name 13) (tile <o13> ↑name <v13>)
  (binding <x21> ↑cell <c21> ↑tile <o21>)
  (cell <c21> ↑name 21) (tile <o21> ↑name <v21>)
  (binding <x22> ↑cell <c22> ↑tile <o22>)
  (cell <c22> ↑name 22) (tile <o22> ↑name <v22>)
  (binding <x23> ↑cell <c23> ↑tile <o23>)
  (cell <c23> ↑name 23) (tile <o23> ↑name <v23>)
  (binding <x31> ↑cell <c31> ↑tile <o31>)
  (cell <c31> ↑name 31) (tile <o31> ↑name <v31>)
  (binding <x32> ↑cell <c32> ↑tile <o32>)
  (cell <c32> ↑name 32) (tile <o32> ↑name <v32>)
  (binding <x33> ↑cell <c33> ↑tile <o33>)
  (cell <c33> ↑name 33) (tile <o33> ↑name <v33>)
  -->
  (tabstop <tab>)
  (write2 (crlf) (tabto <tab>) <name> "(" <s> ") --> " <n> (crlf))
  (write1 (tabto <tab>) " -------------" (crlf))
  (write1 (tabto <tab>) " |" <v11> "|" <v21> "|" <v31> "|" (crlf))
  (write1 (tabto <tab>) " |---|---|---|" (crlf))
  (write1 (tabto <tab>) " |" <v12> "|" <v22> "|" <v32> "|" (crlf))
  (write1 (tabto <tab>) " |---|---|---|" (crlf))
  (write1 (tabto <tab>) " |" <v13> "|" <v23> "|" <v33> "|" (crlf))
  (write1 (tabto <tab>) " -------------" (crlf)))
```

8.8. Set-up

Once all the productions and the representations have been defined, a few house-keeping operations need to be performed. These should be included at the beginning of the file that contains the productions that define the task.

8.8.1. Multi-attributes

To improve the ordering of productions, the function **multi-attributes** is called with a list of those classes that have attributes with more than one occurrence per object and, if known, the number of occurrences. In this implementation of the Eight Puzzle, states and desired states have multiple bindings, and cells have links to other cells.

```
(multi-attributes  '((state binding 9) (desired binding 9)
   (cell cell 4)))
```

8.8.2. Trace-attributes

The user can improve the readability of a trace by providing a list of attributes to be traced for different classes. In the Eight Puzzle, the operators do not have distinguishing names, so the only way to obtain a meaningful trace of the problem solving is to include the cell of the operator in trace-attributes. The cell of the operator contains the position of the tile that is moved into the blank.

```
(trace-attributes '((operator tile-cell)))
```

8.9. Search Control

Besides defining the task (the goal and the problem space), additional search control can be introduced to make problem solving more efficient.

8.9.1. Simple Search Control

Eight*worst-undo creates a worst-preference for the inverse of the operator that created the current state. This type of search control is common and many tasks will have productions similar to this one. The key part of the production is the determination of the inverse of an operator. In the Eight Puzzle, the inverse of the prior operator is determined by finding the operator that will move the tile that was moved by the prior operator.

```
(sp eight*worst-undo
  (gc <g> ↑problem-space <p> ↑state <s>)
  (problem-space <p> ↑name eight-puzzle)
  (state <s> ↑moved-tile-binding <mtb>)
  (binding <mtb> ↑cell <cmtb>)
  (preference <o> ↑role operator ↑value acceptable
       ↑problem-space <p> ↑state <s>)
  (operator <o> ↑tile-cell <cmtb>)
  -->
  (preference <o> ↑role operator ↑value worst
        ↑goal <g> ↑problem-space <p> ↑state <s>))
```

8.9.2. Using State Evaluations

State evaluations are a standard way of controlling search. A production that computes the evaluation should look like the following. (Everything in bold should be left alone. Everything in regular font should be replaced for the specific task.)

```
(sp production-name
        (gc <g> ↑problem-space <p> ↑state { <> <ss> <s> }
                ↑superoperator <so>)
        (problem-space <p> ↑name task-problem-space-name)
        (operator <so> ↑name evaluate-object ↑evaluation <e>
                ↑superstate <ss> ↑desired <d>)
        ; Conditions that compute the evaluation based on state <s> and
        ; desired state <d>. <d> will point to the desired state
        ; defined at the beginning of the task and attached to the
        ; desired and desired roles of the top goal.
   -->
   (evaluation <e> ↑numeric-value your-evaluation))
```

The default productions take care of the rest, testing the supergoal and comparing the evaluations (if **<d>** is augmented with **↑better higher/lower**). A complete evaluation production for Eight Puzzle is below. It gives an evaluation of 1 if the operator that created the state moved a tile into its desired position. A second production gives an evaluation of -1 if a tile is moved out of position, and a third production gives an evaluation of 0, if neither of these occur.

```
(sp eight*eval-state-plus-one
   (gc <g> ↑problem-space <p> ↑state { <> <ss> <s> }
        ↑superoperator <so>)
   (problem-space <p> ↑name eight-puzzle)
   (operator <so> ↑name evaluate-object ↑evaluation <e>
        ↑superstate <ss> ↑desired <d>)
   (state <s> ↑moved-tile-binding <b1>)
   (binding <b1> ↑cell <c1> ↑tile <v1>)
   (desired <d> ↑binding <b2>)
   (binding <b2> ↑cell <c1> ↑tile <v1>)
   -->
   (evaluation <e> ↑numeric-value 1))
```

8.10. Example Trace

After loading all of the Eight Puzzle productions into *Soar*, it is ready to run. Below is a trace of the problem solving and learning for the Eight Puzzle. All output is shown in boldface. (The trace of the initial and desired states at the beginning was not produced by the program.) All comments are prefaced by a semi-colon (;).

```
(soarload 'eight.soar)
(learn on full-trace)
(d 12)
learn status: on always print full-trace
0    g: g0001
   initial state            desired state
-------------          -------------
| 2 | 8 | 3 |          | 1 | 2 | 3 |
|---|---|---|          |---|---|---|
| 1 | 6 | 4 |          | 8 |   | 4 |
|---|---|---|          |---|---|---|
| 7 |   | 5 |          | 7 | 6 | 5 |
-------------          -------------
1    p: p0004 eight-puzzle
2    s: s0005
3    ==>g: g0002 (tie operator undecided)
4       p: p0051 selection
5       s: s0053
6       o: o0056 evaluate-object(move-tile(13))
7       ==>g: g0045 (no-change operator evaluate-object(move-tile(13)))
8          p: p0004 eight-puzzle
9          s: s0005
10         o: o0042 move-tile(13)
             -------------
             | 2 | 8 | 3 |
             |---|---|---|
             | 1 | 6 | 4 |
             |---|---|---|
             |   | 7 | 5 |
             -------------
11         s: s0058
; An evaluation of -1 is created for s0058 because the 7 was
; moved out of its desired position.  This evaluation leads to
; the termination of goal g0045 and will be followed by the
; evaluation of another eight-puzzle operator.
;
; Since learning is on, a chunk will be built.  Below is a trace of
; the production being built.  This trace is produced because of
; full-trace learning.
```

```
backtracing to determine conditions
working-memory elements that will become actions:
(evaluation e0057 ↑numeric-value -1)
productions and conditions traced through:
eight*eval-state-minus-one
  decision-procedure
     eval*select-role-operator
        (gc g0002 ↑operator o0056)
        (operator o0056 ↑name evaluate-object)
        (operator o0056 ↑role operator)
        decision-procedure
        (operator o0056 ↑object o0042)
        (operator o0056 ↑superproblem-space p0004)
        (operator o0056 ↑superstate s0005)
        (operator o0056 ↑desired d0003)
  (problem-space p0004 ↑name eight-puzzle)
  decision-procedure
     eight-create-new-state
        decision-procedure
        decision-procedure
        (state s0005 ↑blank-binding b0025)
        (operator o0042 ↑tile-cell c0020)
        (operator o0042 ↑name move-tile)
        (state s0005 ↑binding b0019)
        (binding b0019 ↑cell c0020)
        (state s0005 ↑binding b0025)
        (binding b0025 ↑cell c0026)
        (binding b0025 ↑tile t0006)
        (binding b0019 ↑tile t0013)
  (cell c0020 ↑cell c0026)
  (desired d0003 ↑binding d0035)
  (binding d0035 ↑cell c0020)
  (binding d0035 ↑tile t0013)
  (operator o0056 ↑evaluation e0057)
conditions that are tersed out: (binding <b1> ↑tile <t2>)

build:p0086
12     o: o0054 evaluate-object(move-tile(22))
***break***
```

(last-chunk)

; Print out the production that was just built.

```
(sp p0086
    (gc <g1> ↑operator <o1>)
    (operator <o1> ↑role operator ↑name evaluate-object
        ↑superproblem-space <p1> ↑object <o1> ↑superstate <s1>
        ↑desired <d2> ↑evaluation <e1>)
    (problem-space <p1> ↑name eight-puzzle)
    (state <s1> ↑blank-binding <b1> ↑binding <b2>
        ↑binding { <> <b2> <b1> })
    (operator <o1> ↑name move-tile ↑tile-cell <c1>)
    (cell <c1> ↑cell <c2>)
    (binding <b2> ↑cell <c1> ↑tile <t1>)
    (binding <b1> ↑cell <c2>)
    (desired <d2> ↑binding <d1>)
```

```
    (binding <d1> ^tile <t1> ^cell <c1>)
  -->
    (evaluation <e1> ^numeric-value -1))
nil
(learn trace)

learn status: on always print trace t
; Disable the trace of the production construction.

(run 7 d)

13      ==>g: g0046 (no-change operator evaluate-object(move-tile(22)))
14         p: p0004 eight-puzzle
15         s: s0005
16         o: o0044 move-tile(22)
             -------------
             | 2 | 8 | 3 |
             |---|---|---|
             | 1 |   | 4 |
             |---|---|---|
             | 7 | 6 | 5 |
             -------------
17         s: s0065
build:p0087
; This has an evaluation of 1 because the 6 was moved into
; its desired cell.
18      o: o0055 evaluate-object(move-tile(33))
18:42 p0086
; The chunk built for the first subgoal applies and computes an
; evaluation of -1 because the 5 tile will be moved out of its desired
; cell by operator o0043.
;
; Once all the evaluations are computed, preferences are created
; that compare the different operators based on their evaluations.
; Two of the evaluations are the same, so indifferent preferences
; are created between operators o0043 and o0042.  Both of these
; are worse than o0044, so worse-preferences are also created.
; These preferences are the results of g0002, the goal with the
; tie impasse.
; Since the problem solving to create the two worse-preferences
; was identical, two identical chunks could have been built.
; The duplication is detected (although it is not always detected)
; and only one production is built.  Duplicate chunks are also built
; because of the symmetry in the productions that create the
; indifferent-preferences.
; The productions are refracted so that they do not fire
; on the data that was used to create them.

duplicate chunk
build:p0088
duplicate chunk
build:p0090

19   o: o0044 move-tile(22)
***break***
```

```
(last-chunk)

(sp p0090
    (gc <g1> ↑desired <d3> ↑state <s1> ↑problem-space <p1>)
    (problem-space <p1> ↑name eight-puzzle)
    (state <s1> ↑blank-binding <b2> ↑binding <b2>
        ↑binding { <> <b2> <b1> } ↑binding { <> <b1> <> <b2> <b3> })
    (binding <b2> ↑cell <c3>)
    (binding <b1> ↑cell { <> <c3> <c1> } ↑tile <t2>)
    (cell <c1> ↑cell <c3>)
    (desired <d3> ↑binding <d1> ↑binding { <> <d1> <d2> })
    (binding <d1> ↑cell <c1> ↑tile <t2>)
    (binding <b3> ↑cell { <> <c1> <> <c3> <c2> }
        ↑tile { <> <t2> <t3> })
    (cell <c2> ↑cell <c3>)
    (binding <d2> ↑cell <c2> ↑tile <t3>)
    (preference <o2> ↑role operator ↑value acceptable
        ↑goal <g1> ↑problem-space <p1> ↑state <s1>)
    (operator <o2> ↑name move-tile ↑tile-cell <c2>)
    (preference <o1> ↑role operator ↑value acceptable
        ↑goal <g1> ↑problem-space <p1> ↑state <s1>)
    (operator <o1> ↑cell <c1>)
  -->
    (preference <o2> ↑role operator ↑value indifferent ↑reference <o1>
        ↑goal <g1> ↑problem-space <p1> ↑state <s1>))
nil
75:(run)

20  s: s0078
     -------------
     | 2 | 8 | 3 |
     |---|---|---|
     | 1 |   | 4 |
     |---|---|---|
     | 7 | 6 | 5 |
     -------------
21:48 p0090
21:48 p0090
; Chunk p0090 detects that moving the 4 and moving the 6
; are indifferent because they both move a tile out of its
; desired cell. This does not determine the next operator
; so a tie impasse is created.
21  ==>g: g0047 (tie operator undecided)
22     p: p0085 selection
```

This continues until the problem is solved.

9. Advanced Topics

9.1. Operator Implementation Goal Tests

If an operator requires a subgoal to implement it and some test exists to determine if a state is a valid result, recent work suggests unusual way to structure the subgoal. The advantage of this scheme is that even if over-general chunks are learned, they will not screw things up. The disadvantage is that the chunks will often return multiple states. In the subgoal for implementing the operator (call it Op1), there should be a production (call it **detect-candidate**) that detects that a state is a *candidate* result. A candidate result is a state that might be a valid result of the subgoal although the final test has not been made. It is possible that all states in the subgoal are candidate results. It is also possible that the candidate result is not a state in the subgoal, but only a subobject (or whatever). The action of **detect-candidate** is to augment the superstate (the superstate is the state that Op1 is being applied to) with an object of class *result*. The result will be augmented with the operator (Op1) and the candidate result. For example, **detect-candidate** might be:

```
(sp detect-candidate
  (gc <g> ↑problem-space <p> ↑state <s>
        ↑supergoal <sg> ↑superoperator <so>)
  (problem-space <p> ↑name implement-op1)
  (state <s> ↑candidate yes) ;some test that it is a candidate
  (operator <so> ↑name op1)
  (gc <sg> ↑state <ss>)
  -->
  (state <ss> ↑result <r>)
  (result <r> ↑operator <so> ↑candidate <s>))
```

In most *Soar* programs, this production would have just created the preference for the state in the supercontext and the subgoal would terminate. In this scheme, a second production, call it **detect-op1-success**, will create the preference. This preference will fire *outside* the subgoal so that it will not be included in the chunk. For example:

```
(sp detect-op1-success
  (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  (problem-space <p> ↑name xyzzy)
  (state <s> ↑result <r>)
  (operator <o> ↑name op1)
  (result <r> ↑operator <o> ↑candidate <c>))
  (state <c> ↑attribute value)     ;some test that it is a valid state
  -->
  (preference <c> ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>))
```

This production will fire whenever a candidate has been suggested that passes the final test. When chunking is used, the chunk will have as its actions all states that were candidates (but no preferences). **Detect-op1-success** will select out the correct result and create a preference. If the chunk applies incorrectly, **detect-op1-success** will not fire and the subgoal will be used.

9.2. Operator Parallelism

The parallel-preference allows the user to specify that two or more operators can be performed in parallel. In the decision procedure, if the result is a set of operators that are mutually parallel (there exist parallel preferences between them), the current goal-context-info for the operator role is removed; and new operator goal-context-infos are created for each of the parallel operators. Whenever a parallel operator is rejected, its goal-context-info is removed from working memory. The parallel structure is maintained until a new preference causes a change in a higher-order object or all but one of the parallel operators is rejected. Each parallel operator is independent and each can cause productions to fire independently of the others. If a parallel operator does not lead to the creation of a preference that will change the context, a no-change impasse will arise. To distinguish the subgoals, each has a **↑superoperator** augmentation that contains the identifier of one of the parallel operators. When the operators have subgoals, they will run in parallel. These subgoals can also have parallel operators, giving rise to exponential blowups in the number of subgoals being pursued (making the goal-context-stack really a tree). The parallelism is only simulated in the present implementation. All of the parallel operator subgoals are synchronized on the decision cycle. The function pgs will print the parallel structure and make a little more sense of it than the trace (see Section 10.5.2).

This simple parallel structure gives AND, OR and hybrid AND-OR parallelism. If all of the operators are non-monotonic (they all create new states), we have OR parallelism where all of the parallel operators are racing to succeed first. If two (or more) parallel operators finish on the same decision cycle, there will be two (or more) acceptable-preferences for the states, and this will lead to a tie impasse if no other preferences are added. Eventually one of these willed be picked after going into the selection problem space .

If all of the operators are monotonic and just add information to the current state until enough information is available to make a new decision, we have AND parallelism. A good example of this is when parallelism is applied to the evaluate-object operators in the selection problem space (see Section 6.3). In parallel, all objects will be evaluated until enough evaluations and preferences are created to break the tie that created the selection subgoal. If there is a combination of monotonic and non-monotonic operators, we get a hybrid AND-OR parallelism, where the monotonic operators augment the current state until a non-monotonic operator terminates.

Since all parallel operators are running in the same working memory it is possible for them to share information and to communicate partial results. One way to achieve this is to have the operators attach partial results to the state they are applying to and examine the state for information created by other operators.

10. Top-level Variables and Functions

This chapter consists of the global variables, properties, and functions that are used to control ***Soar***. Some of these are ***Ops5*** commands that have been modified to provide more functionality. The **Backup** feature of ***Ops5*** does not work in ***Soar*** (but see **pop-goal** for a reasonable alternative). The functions names are followed by a list of their arguments. Arguments in square brackets ([]) are optional. An argument ending in * signifies that any number of arguments may follow.

10.1. Global Variables

The following global variables are used to control certain aspects of ***Soar***. Many of these are also referred to in sections on functions that they affect. All global variables in ***Soar*** begin and end with an asterisk (*).

chunk-all-paths If T, then when the exact same subgoal result is produced by two or more production firings, chunks will be built based on each of the production firings. ***Chunk-all-paths*** is initially **nil**.

chunk-classes A list of SP class names for which at least one must occur in the conditions of a chunk for it to be built. This helps eliminate chunks that are overly general. ***Chunk-classes*** is initially **(problem-space state operator)**.

chunk-free-problem-spaces A list of problem-space names for which chunking should not be used. If the current problem space in a subgoal has its name in the list, and the subgoal is terminated, no chunk will be built for that subgoal.

chunks A list of the names of all of the productions that have been learned.

max-chunk-conditions No production will be built that has a greater number of conditions than ***max-chunk-conditions***. ***Max-chunk-conditions*** is initially 200.

max-elaborations If the elaboration phase runs more that ***max-elaborations*** then the elaboration phase is terminated and the decision procedure is executed. The default value of ***max-elaborations*** is 100.

max-recurse The maximum recursive depth that the ordering algorithm will use in breaking ties between competing conditions. By increasing the depth, the ordered productions can sometimes be more efficient, although loading in the productions will take longer. ***Max-recurse*** is initially 2.

sp-classes A list of dotted pairs where the first element of each dotted pair is the SP class name and the second element is the P class name. When translating from SP format, ***Soar*** uses ***sp-classes*** to replace SP classes with P classes. Users should not have to add pairs to ***sp-classes***, since this is done automatically by ***Soar***. The first time a SP class is encountered, it, along with its name concatenated with **-info** is added to ***sp-classes***. The user should add pairs to ***sp-classes*** if he wants to have more than one SP class

translated into the same P class (gc, goal-context, context, and goal all translate into goal-context-info).

spo-default-depth The default depth of objects that spo prints out. The value of ***spo-default-depth*** is initially 1.

subgoal-tabs If T, **watch** and **pgs** will indent during the tracing or printing of the context stack. If **nil**, **watch** and **pgs** will not indent, but instead will print the subgoal depth as a number. The value of ***subgoal-tabs*** is initially T.

warning If T, warnings are printed. If **nil**, warnings are not printed. The value of ***warning*** is initially T.

watch-free-problem-spaces Contains a list of problem-space names that will not be traced with watch 0. The value of ***watch-free-problem-spaces*** is initially **nil**.

10.2. Initialization

10.2.1. Init-soar

While running ***Soar***, the user may wish to empty working memory and restart a run using the same core image. The function init-soar empties working memory. It should be called whenever the user wishes to restart without reloading productions. After it has been called, new productions can be added, either manually or by reading a file. Old productions (including chunks), that haven't been replaced, will still be available.

```
(init-soar)
```

10.2.2. Restart-soar

While running ***Soar***, the user may wish to replace all of the productions, but still maintain the same ***Lisp*** core image. The restart-soar function is a ***Soar*** function that re-initializes the system, removes all productions, including chunks, empties working memory and resets all global variables to their initial (default) values.

```
(restart-soar)
```

10.2.3. Init-context *id1 id2 id3*

The init-context function first calls init-soar to clear working memory, and then creates the context in working memory. If it is not called, the initial context, except for the goal, is all **undecided**: **(gc g0001 ↑problem-space undecided ↑state undecided ↑operator undecided)**. There are three arguments. The first is the identifier of the initial problem-space, the second is the identifier of the initial state and the third is the identifier of the initial operator. The function **gensyms** a goal identifier, which is returned as the result.

```
(init-context 'problem-space1 'state1 'do-eight-puzzle)
```

10.3. Loading, Running, and Breaking

10.3.1. Soarload *file*

The soarload function will load in file *file*. It must be used in place of load on Xerox D-machines for files containing productions, but its use is optional for all other implementations of ***Soar***.

```
(soarload 'eight.soar)
```

10.3.2. Multi-attributes *L*

The multi-attributes function takes a list of two- or three-element lists as its argument. The first element of each sublist is a SP class name, the second element is a SP attribute (not an ***Ops5*** attribute, but the attributes that show up in SP format), and the third (optional) element is a number. The function declares that the attribute for the SP class will appear multiple times for a given object. This usually happens when an object has a set of subobjects. The third argument is the expected number of occurrences of the attribute for a given object of that class. The default is 5. When this information is provided, the ordering algorithm can produce more efficient P format productions and greatly speed up the execution of a system.

10.3.3. Run *N* [*D*]

The run function executes the production system with the current working memory for the number of cycles given by *N*. If *D* is missing, *N* gives the number of *production* cycles to be executed. In ***Soar***, during the elaboration phase, many productions may fire in parallel on the same production cycle. This is one production cycle. However, the elaboration phase may last many production cycles, and each cycle is counted toward the total. Each decision phase is also counted as one production cycle. If *D* is **d** (no other values are legal), then *N* is the number of *decision* cycles that are executed before halting. In this case ***Soar*** halts just after the decision procedure of the *N*th decision cycle. If *N* is an object identifier or object name, ***Soar*** halts when an object with that identifier or name is selected as the current value of a role in a context. If *N* is a SP-form working-memory element, ***Soar*** halts when that working-memory element is created. If a run is done following init-soar, it automatically initializes working memory with all non-goal roles in a goal-context being **undecided**.

```
(run 100 d)
```

10.3.4. D *N*

(D *N*) is equivalent to (Run *N* D).

10.3.5. Pbreak *L**

Pbreak allows the user to give a set of names of productions, and break on the production cycle *after* they fire. It has been expanded in ***Soar*** to allow the user to break after an object with a specific name is selected for a context role. *L* can either be the name of a production to break after, or it can be a name or identifier of the object the user wishes to break on. ***Soar*** will break following the decision procedure when an object with that name or identifier is selected as current. If *L* is **nil**, all break points are listed.

```
(pbreak selection evaluate-object)
(pbreak initialize-r1-problem-space reject-worse)
```

10.3.6. Unpbreak *L**

Unpbreak removes breaks set by pbreak. To remove a break, use the same argument in unpbreak as was used in pbreak. If *L* is **nil**, all breaks are removed.

```
(unpbreak nil)
(unpbreak initialize-r1-problem-space reject-worse)
```

10.3.7. User-select *X*

If *X* is T, then whenever ***Soar*** is going to make a choice between indifferent objects, the user will be asked to make the selection. If *X* is **nil**, ***Soar*** will make the selection randomly. If *X* is **'first**, ***Soar*** will always select the first one found. This is a deterministic selection. If *X* is a list, then the list should contain numbers or atoms. For each selection, the first element of the list is stripped off and used to select an object. If it is a number, it will be used to index into the list of objects to be selected (1 for the first). If the number is less than 1, or greater than the total number of choices, the user is asked. If it is a symbol, the objects are examined, and the first one that has the symbol as a name or the value of a trace-attribute is selected. If the symbol does not match any of the choices, the user is asked. When the list is exhausted, user-select is called automatically with the value of ***default-user-select***, which is initially T. The original value for user-select is **'first**.

```
(user-select t)
```

10.4. Tracing

10.4.1. Trace-attributes *L*

Trace-attributes takes a list of two-element lists as its argument. The first element of each sublist should be a SP class and the second element should be a SP attribute. After trace-attributes is called, a watch trace of level 0-2 (and PGS) will print out the value of the specified attributes when an object is selected to a context role. If the value is an identifier with a **↑name** attribute, then the name of the identifier is printed. The tracing is recursive, so that if the value is an identifier that appears in an augmentation with another class in trace-attributes, its attributes will be traced, and so on. The recursion stops whenever a previously traced identifier, or one that has no trace-attributes, is encountered. Trace-attributes is initialized with ((goal role) (goal impasse) (goal superoperator) (operator instance) (operator object)). The **↑name** attribute is handled specially for all classes, so it should not be included in trace-attributes. All calls to trace-attributes merely add pairs to the list.

```
(trace-attributes '((state backplane) (operator module) (module size))
```

10.4.2. Watch *N*

As in ***Ops5***, *N* is a parameter that determines the amount of trace information produced by the system. ***Soar*** expands the available values and expands the different levels of trace information.

-1 — No tracing.

0 — Object tracing. Changes to a goal-context are listed. No production or working memory tracing. The object tracing includes the current decision cycle number, the role being changed, the identifier of the object, the name and any attributes declared with trace-attributes (see above). Objects are indented (3 * the subgoal depth). Indenting can be turned off by setting the global variable ***subgoal-tabs*** to **nil**. When there is no indenting, the subgoal depth is printed at the beginning of each line. Subgoals are prefaced by "==>" so they are easy to pick out.

```
1 ==>g: g0001 (no-change goal)
2     p: p0003 eight-puzzle
3     s: s0012
4     ==>g: g0031 (tie operator)
5        p: p0032 selection
6        s: s0033
7        o: o0036 evaluate-object(up)
```

.5 — Same as 1, except no trace of the time-tags of working-memory elements that match the conditions of the productions, or are created by productions or are auto-removed.

1 — Adds trace of the productions that fire. In ***Soar***, the trace starts with the decision cycle number followed by the *production* cycle number (the number of production cycles — where many productions can fire in parallel on one production cycle — since the last init-soar). These numbers are followed by the name of the production that fired.

When the decision procedure is executed, the role and the name of the selected object are traced. If there is an impasse in the decision procedure, the type of impasse and the name of the newly created subgoal is printed. Following this information is a list of the data that was matched by the production (given by time-tags) followed by the data that was created by the production (given by time-tags). These working-memory elements are *Ops5* working-memory elements and will not be in SP format if printed out directly using the wm function. For example:

```
73:174 decide operator s0415
       o: up 1466
74:175 create-newstate 1443 1456 17 1463 1466 23 --> 1467
```

The first line is a trace of a decision occurring during the 73rd decision cycle. It is the 174th production cycle and operator S0415 (also called **up**) is selected. 1466 is the time-tag of the working-memory element for the current operator. On the following production cycle, production **create-newstate** fires using the six working-memory elements listed to create 1467. On the return from subgoals, the working-memory elements that were garbage-collected are listed following "<--".

1.5 Just like 1, except that the actual working-memory elements added to and removed from working memory are printed.

2 Prints out the time-tags of the working-memory elements matched by the conditions of the production and the actual working-memory elements added to and removed from working memory.

Default = 0.

Example:

```
(watch 1)
```

10.4.3. Decide-trace *X*

If *X* is T, decide-trace is enabled. If *X* is **nil**, decide-trace is disabled. The default is **nil**. When decide-trace is enabled, a trace of the decision procedure is displayed.

```
(decide-trace nil)
```

10.4.4. Ptrace *X*

If *X* is a production name, it will be traced whenever it fires. If *X* is an SP-form working-memory element, that working-memory element is traced when it is created or matched by a firing production. If *X* is an object name or identifier, all working-memory elements that augment that object are traced when they are created or matched by a firing production. Tracing of chunks is also controlled by the **trace** option of learn.

```
(ptrace create-new-state)
```

10.4.5. Unptrace

Removes traces set by ptrace.

```
(unptrace)
```

10.5. Displaying Information

10.5.1. CS

The cs function produces a listing of the productions that are in the conflict set. In ***Soar***, these are the productions that will fire on the next production cycle. If the next cycle is an elaboration phase, the elaboration productions that will fire are displayed. If the next production cycle is a decision, the number of instantiations of **decision*gather-preferences** is displayed. **Decision*gather-preferences** matches all of the preferences relevant to the context stack. Note: some elaboration productions may be in the conflict set but not change working memory because the elements they create are already in working memory.

```
(cs)
```

10.5.2. PGS

This prints out the goal-context stack, indented at each subgoal, followed by the decision cycle number. If ***subgoal-tabs*** is **nil**, the indentation will be replaced by numbered depth counts. For parallel operators, the goal stack is printed out depth-first, with a space between the end of one parallel operator's subgoal tree and the beginning of the next parallel operator. This is a great function for finding out where you are in problem solving.

```
(pgs)
```

10.5.3. SPR *X**

The spr function is the generic SP printer for all types of objects. It takes any number of arguments which can be time-tags, object identifiers, partial descriptions or production names. It then prints the associated working memory elements or productions appropriately. If no argument is given, it calls pgs.

```
(spr (operator ^name evaluate-object))
```

10.5.4. PPWM *X**

Without any arguments, ppwm prints out all of working memory. Arguments to ppwm provide a partial description of working-memory elements in P-format: a class and attribute-value pairs. These arguments act as a filter, so that only those working-memory elements that match are printed. In the example, the second call will print out only acceptable-preferences for goals.

```
(ppwm)
(ppwm preference ^role goal ^value acceptable)
```

10.5.5. SPPWM *X**

The sppwm function is an SP version of ppwm. Its input is a partial description of an object in SP format. It finds all objects matching that description and prints them in SP format.

```
(sppwm operator *name evaluate-object)
```

10.5.6. WM *N**

The wm function takes any number of time-tags as its argument, and prints out the working-memory elements with those time-tags. The time-tags of working-memory objects are listed when they are created during watch 1 and 2.

```
(wm 45 54)
```

10.5.7. SWM *N**

The swm function takes any number of time-tags as its argument, and prints out the objects with the identifiers of working-memory elements with those time-tags. The time-tags of working memory objects are listed when they are created during watch 1 and 2.

```
(swm 45 54)
```

10.5.8. PO *I*

The po function will print out the augmentations of the object with identifier *I* (it only accepts one argument at a time). This will print out preferences and augmentations where the object is in the identifier field. It will not print out your own weird data structures if *identifier* is not in the identifier field.

```
(po S0003)
```

10.5.9. SPO *I** [*D*]

The spo function is an expanded SP version of po. It prints out the augmentations of the identifiers in SP format. It does not print out preferences. It has an optional final argument: *depth*. If *depth* is given, spo will print out a depth-first expansion of the objects and subobjects to depth *D*. It will only print the augmentations of each object once. The default depth (for when no second argument is provided) is held in global variable ***spo-default-depth***, which is initially 1.

```
(spo S0003 2)
```

10.5.10. SPOP *I** [*D*]

The spop function will print out the preferences of the identifiers in SP format. It does not print out augmentations. It has an optional final argument: *D*. If *D* is given, spo will print out a depth-first expansion of the preferences of objects in the context fields of preferences of each object once. The default depth (for when no second argument is provided) is held in global variable ***spo-default-depth***, which is initially 1.

```
(spop S0003 2)
```

10.5.11. PM *P**

The pm function prints out production *P* in P format.

```
(pm eight*create-new-state)
```

10.5.12. SPM *P**

The spm function prints out production *P* in SP format.

```
(spm eight*create-new-state)
```

10.5.13. Matches *P**

The matches function lists the time-tags for all of the working-memory elements that match the conditions of production *P*. It also prints all of the partial instantiations of production *P* (with time-tags).

```
(matches eight*create-new-state)
```

10.5.14. Smatches *P**

The smatches function takes the name of a production as its argument (unquoted). It prints out the most complete match for the production given the current working memory (as time-tags) followed by a listing of the production with a pointer to the condition where the match failed. Each condition in the production, is prefaced by the number of partial instantiations active at that point. This function subsumes most of the interesting aspects of matches.

```
(smatches eight*create-new-state)
```

10.5.15. Back-trace [*I*] [*G*]

The back-trace function lists all the productions used in goal *G* to produce the working-memory elements described by *I*. It also prints out the working-memory elements that were matched by those productions that would be included in a chunk if it were to be built with *I* as its actions. If *G* is not provided, the most recent subgoal is used. *I* can be either a time-tag of a working-memory element, an object identifier (in which case all augmentations of the object are used), or a SP pattern that includes at least one attribute (in which case all

working-memory elements matching the SP pattern are used). If *I* is not included, back-trace will use the actions for goal *G* (if there are no actions at this time, nothing will be printed).

Beginning with the working-memory elements described by *I*, the productions that created *I* are found, their names are printed, and the working-memory elements that matched their conditions are collected. If the working-memory element was created in a subgoal, the working-memory elements that would be used as conditions for a chunk for that subgoal are collected, and the identifier of the subgoal is printed. Printing from then on is indented until all the collected working-memory elements have been processed. If a working-memory element is the same as a working-memory element that has already been processed, it is ignored. If a collected working-memory element was created before *G*, it is printed because it will be the basis of a condition in a chunk built for *G*. If a collected working-memory element was created by another production firing in the subgoal, or by a subgoal, or by the decision procedure, then the process recurses. If a collected working-memory element was created by the decision procedure (either a context slot or a goal augmentation) **decision-procedure** is printed and the working-memory element associated with that creation act is back-traced (see Section 7.1 for more information).

```
(back-trace o0034)
(back-trace (evaluation e0021 ↑numeric-value -1) g0032)
```

10.5.16. PI *P* [*N*]

The pi function prints out the working-memory elements that form the *N*th partial instantiation for production *P*. If *N* is missing, the first partial instantiation is listed.

```
(pi eight*create-new-state)
```

10.5.17. Print-stats

The print-stats function lists a summary of statistics for the runs of ***Soar*** since start-up or the last call to init-soar. Most of the statistics concern a set of events, such as production firings, decision cycles, etc. The total number of each type of event is given, along with the number of events per second.

- *Number of productions:* The is the total number of productions in the system, including all chunks built during problem solving.
- *Number of nodes, with sharing/without sharing:* The first number is the number of nodes actually used in the network. The second number is the number of nodes that would be required if there were no sharing.
- *Elapsed time:* On a Vax or D-machine, this is CPU time. On the 3600 this is elapsed real-time while running.
- *Number of decision cycles:* This is the total number of decision cycles.

- *Number of production cycles:* This is the total number of production cycles that were executed, which include the number of decision cycles and elaboration cycles. This is not the total number of production firings, since elaborations fire in parallel.

- *Number of elaboration cycles per decision cycle:* This is the average number of elaboration cycles executed during a decision cycle. This is computed by computing the total number of elaboration cycles (production cycles - decision cycles) and dividing by the number of decision cycles.

- *Number of production firings:* This is the total number of productions that were fired. Each decision cycle is counted as one and only one production firing.

- *Number of elaboration productions firing in parallel:* This is computed by dividing the number of elaboration production firings (total production firings - decision cycles) by the number of elaboration cycles.

- *Number of actions:* This is the total number of actions. This includes all additions and deletions from working memory.

- *Working memory size:* This gives the average, total, and current number of working-memory elements.

- *Token memory size:* This gives the average, total, and current number of tokens used to represent the working-memory elements in the RETE network. When this number is large, the system tends to slow down.

Below is an example from running the Eight Puzzle.

```
(print-stats)

Run Statistics
69 Productions (1034 // 3329 Nodes)
21 Seconds Elapsed
22 Decision Cycles (1.047619  per sec.)
47 Prod Cycles (2.238095  per sec.)
        (1.136364  E cycles/ D cycle)
112 Prod Firings (5.333334  per sec.)
        (3.6  Elab. prod. in parallel)
498 RHS Actions (23.71429  Per Sec.)
191 Mean working memory size (260  Maximum  222  Current)
419 Mean token memory size (651  Maximum  521  Current)
```

10.6. Changing Working Memory and Production Memory

10.6.1. Make

The make function adds to working-memory the P-format working-memory element that follows it in the function call.

```
(make state-info ↑identifier S4404 ↑attribute name ↑value cleveland)
```

10.6.2. Smake

The smake function adds to working-memory the SP-format working-memory elements that follow it in the function call.

```
(smake state S4404 ↑name cleveland)
```

10.6.3. Sremove *N*

The sremove function removes from working memory the element with time-tag *N*. This can be used only at the top-level to remove working-memory elements and can not be included in production actions. In most *Ops5* implementations, this is just remove, however to avoid confusion with some *Lisp* commands, we call it sremove.

```
(sremove 45)
```

10.6.4. Pop-goal [*X*]

The pop-goal function removes the goal *X*, all its subgoals, and all working-memory elements created in it or its subgoals. No chunks are created when the goal is popped. If *X* is not specified, the last subgoal created is popped. It takes any number of subgoals as arguments, and will pop all of them, however, this is only useful when parallelism is being used. This function allows a limited form of back up in *Soar*. After pop-goal has been executed, *Soar* is in an elaboration phase, and unless the user adds productions or working-memory elements, *Soar* will create a new subgoal in the next decision that is just like the one that was popped

```
(pop-goal g0043)
```

10.6.5. P

The p function creates a P format production. If this replaces a previously created production (same name, different body) the old production is excised and the name of the excised production is printed.

```
(p eight*create-new-state elaborate
        (goal-context-info ↑identifier <g> ↑attribute state ↑value <s>)
        (goal-context-info ↑identifier <g> ↑attribute operator
                ↑value <o>)
        (op-info ↑identifier <o> ↑attribute name ↑value up)
        -->
        (make state-info ↑identifier <n> ↑attribute name ↑value down))
```

10.6.6. SP ...

The sp function creates a SP format production. If this replaces a previously created production (same name, different body) the old production is excised and the name of the excised production is printed.

```
(sp eight*create-new-state
        (gc <g> ↑state <s> ↑operator <o>)
        (operator <o> ↑name up)
        -->
        (state <n> ↑name down))
```

10.6.7. Excise *P*

The excise function removes production *P* from production memory. If a production is excised, a "#" is displayed.

```
(excise eight*create-new-state)
```

10.7. Chunking

10.7.1. Learn [*A*]

This function is called to modify or examine a number of flags that control chunking. The arguments are not evaluated. If no arguments are included, all of the flags are displayed. Below is the list of argument pairs, the first one (underlined) is the default.

- **never/on/off**
  On turns learning on, **off** turns learning off. **Never** turns learning off and learning can not be used before init-soar is called. If learning is **off**, but not **never**, it can be turned on (and off) at anytime during a run. With **never**, ***Soar*** does not maintain the extra information required by the learning mechanism. **Never** runs about 8% faster than **off**, which runs about 25% faster than **on**. These figures depend upon the complexity of the objects in working memory and the frequency of subgoal creation and termination.

- **always/bottom-up**
  With **always**, productions are built whenever a subgoal terminates. With **bottom-up**, productions are only built for terminal subgoals (subgoals that do not have any subgoals).

- **print/noprint/full-print**
  With **print**, production names are printed as they are created. With **noprint**, nothing is printed. With **full-print**, the full production is printed when it is created.

- **trace/untrace/full-trace**
  With **trace**, every time a production is chunked, it is added to a list. When a production on that list fires, it is traced at Watch level 1. With **full-trace**, the building of the production is also traced.

```
(learn on bottom-up full-print)
```

10.7.2. Last-chunk

This will print, in SP format, the last production created by the chunking mechanism.

```
(last-chunk)
```

10.7.3. Excise-chunks

This will excise all productions that have been chunked since starting up *Soar* (either through starting *Soar* or calling restart-soar). The names of all chunked productions are held in ***chunks***. The function uses ***chunks*** to remove the chunked productions and then sets ***chunks*** to **nil**.

```
(excise-chunks)
```

10.7.4. List-chunks

This will print all productions with names in ***chunks*** (whenever a chunk is created, it is automatically added to ***chunks***) in SP format. The chunks are listed in the order they were created.

```
(list-chunks)
```

11. Errors, Warnings, and Recovery Hints

11.1. Errors

- **Illegal production name:** The name of the production was a list.
- **Illegal production type:** The type of the production was neither missing, nor elaborate nor decide.
- **No '-->' in production:** '-->' was not found in the production. This usually arises when there is an extra ')' in the condition elements.
- **Attempt to negate a compound object:** A negation was placed before an SP object that had more than one attribute. This will create a separate working-memory element for each attribute which is not always the desired effect (see Section 3.4). If that is the desired effect, place a negation before each attribute.
- **Didn't find terminator:** A terminator (either >> or }) to match a previously encountered << or { was missing from a condition of the production.
- **Missing >>:** A << is missing a closing >>.
- **Missing }:** A { is missing a closing }.
- **Didn't find a ↑ when expected:** \ - was not followed by a ↑.
- **Atomic conditions are not allowed:** A condition must be a list.
- **Non-numeric constant after numeric predicate.**
- **Wrong context for }:** A } can occur only following a {.
- **Unrecognized symbol.**
- **Not a legal function name.**
- **Condition is too long:** The condition has too many fields. This should never happen.
- **Tab must be a number:** \ unknown P-format field name was encountered.

11.2. Warnings

Miscellaneous

- **Illegal multi-attribute value:** A multi-attribute can only have a range between 0 and 100.
- **Exceeded *max-elaborations*. Proceeding to decision procedure.**

Production syntax

- **Illegal index after ↑.**

- **Constant identifier field in:** An identifier field of an augmentation in a condition must be a variable.

- **Identifier field not constant or variable in:** An identifier field of an augmentation in an action must be a constant or variable.

- **Constant object field in:** An object field of a preference in a condition must not be a constant.

- **Object field not constant or variable in:** An object field of a preference in an action must be a constant or variable.

- **Condition not linked to previous conditions:** The conditions of a production must all be linked to the goal-contexts, either through augmentations or preferences.

Actions

- **Atomic Action:** Actions must be lists.

- **Illegal Action.**

- **Unconnected actions in production:** All variables in the actions of a production must either appear in the conditions or be linked to the conditions through other actions.

- **Illegal decide in production type:** The decide action can only be used in productions of type decide.

- **Illegal make in production type:** The make action can only be used in productions of type elaborate.

- **Illegal remove in Soar production:** The remove action can not be used in productions.

- **Illegal modify in Soar production:** The modify action can not be used in productions.

- **Arguments missing from make action.**

- **Wrong number of arguments for Tabstop.**

- **Illegal argument for Tabstop.**

- **Cannot be called at top level: CALL2.**

- **TABSTOP can not be called at the top level.**

- **Write cannot be called at the top level.**

- **Write: nothing to print.**

- **Writel cannot be called at the top level.**

- Write1: nothing to print.
- Write2: nothing to print.
- Write2 cannot be called at the top level.
- (S)PPWM does not take variables.
- Cannot be called at top level: BIND.
- Bind: Wrong number of arguments to.
- Bind: illegal argument.
- CRLF: Does not take arguments.
- RJUST: Wrong number of arguments.
- RJUST: Illegal value for field width.
- TABTO: Wrong number of arguments.
- TABTO: Illegal column number.

Chunking

- No chunk was built because there were no actions.
- No chunk was built because *max-chunk-conditions* was exceeded.
- No chunk was built because no conditions had a class in *chunk-classes*.

11.3. Recovery Hints

Symptom	Probable cause	Remedy
A Soar rule won't load; it just sits there	Certain syntax errors send the loader into an infinite loop; other times the loader just balks.	Try reloading the rule; also check for syntax errors, such as missing spaces inside curly brackets.
While loading in rules, Lisp tries to evaluate a condition.	There is an extra close parenthesis.	Remove the extra parenthesis.
Two goals are generated followed by a message that Soar must terminate.	1. There are no non-default productions. 2. The initialization production did not fire.	1. Load in productions 2. Make sure it tests (gc <g> ↑supergoal)
Many of the productions just loaded do not fire when they should.	Load was used in Interlisp.	Reload using Soarload.
Soar uses up the *max-elaborations* number of elaboration cycles.	A rule may be producing a wm element which enables the rule to match in a new way, and then produce a new wm element, etc.	Modify the rule so that none of its conditions will match any of its actions.
A rule matches, but is not in the conflict set.	The rule is prevented from firing by refractory inhibition.	A good (but not perfect) indicator of refractory inhibition is when (pi) does not print any wm elements, but just returns a number one greater than the number of conditions in the rule
There is an unexpected tie between the new next state and the initial state.	The preference for the initial state included just the goal and problem space; thus it applies regardless of the state.	Add ↑state undecided to the preference for the initial state.
There is an unexpected tie between the new next state and the state after the initial state.	The preferences from the supergoal are interfering with the subgoal.	Make state preferences sensitive to the goal.

12. Installing Soar

All files for ***Soar*** are available on **h.cs.cmu.edu** in account **/usr/soar**. Each ***Lisp*** dialect has a separate directory that contains all of the files necessary to run ***Soar***. ***Common Lisp***=csoar, ***Franz-Lisp***=fsoar, ***Interlisp***=isoar, and ***Zeta-Lisp***=/soar. Each of these directories include the following files:

read.me	A file that describes how to run this dialect of ***Soar*** and an index of all the files in this directory.
default.soar	The default productions.
eight.soar	The Eight Puzzle productions.
soar.load	A load file that will load in all files necessary to run ***Soar*** except the user files. (This is not necessary for ***Franz-Lisp***.)

To obtain the files via the ARPA-net, send mail either to **soar@h.cs.cmu.edu** or John Laird, Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA. 94304. The information needed to FTP the files will be sent to you. The current method is to login to **h.cs.cmu.edu** under account **ftpguest** with password **cmunix**. However, this procedure is only temporary and may not be supported for very long.

In all systems, the first step in executing ***Soar*** is either loading in files (3600, D-machines, and Suns), executing a core image (***Franz-Lisp***), or executing Lisp with a suspend file (***Common Lisp*** on a Vax). Following this, the default productions and then the task productions should be loaded. In the ***Interlisp*** version, **soarload** should be used in place of **load** when loading ***Soar*** files. At the top-level all systems use the same commands like run, watch, ppwm and print-stats. In the Symbolics 3600, TI Explorer, and Xerox D-machine implementations, hitting any character while ***Soar*** is running will cause it to break at the next production cycle.

13. Performance Comparison

Below is a comparison of the time required to solve a simple problem in the Eight Puzzle on different ***Lisp*** systems in Version 4, release 1. Without learning for the Eight Puzzle it took 143 decisions, 346 production cycles, 660 production firings and 3117 right-hand side actions. All runs were done with a freshly created virtual memory. All times are in seconds. The systems are listed in order of increasing elapsed time. No system specific optimizations were used except that the ***Franz-Lisp*** runs were done with debugging information disabled (although ***Soar*** was developed under ***Interlisp*** so it is more tuned for the Xerox machines). Global variables were declared in all systems. None of the additional declarations that are available in ***Common Lisp*** to enhance efficiency were used. The Sun (run on December 18, 1985) and IBM RTPC (run on January 24, 1986) runs used preliminary compilers. All entries of ?? mean that either the statistic was unavailable or not recorded at the time of the run.

Machine	Software	Physical Memory	Elapsed Time	3600 Ratio	CPU Time	GC Time	Load Factor
Xerox 1132	***Interlisp***	8 Mbytes	127	1.08	127	off	
Symbolics 3600	***Zeta-Lisp***	4 Mbytes	137	1.0	137	off	
Xerox 1132	***Interlisp***	8 Mbytes	149	.92	131	18	
Symbolics 3600	***Zeta-Lisp***	4 Mbytes	153	.90	137	16	
Sun 3	***Common Lisp***	8 Mbytes	176	.78	171	none	
IBM RTPC	***Common Lisp***	4 Mbytes	210	.65	210	??	~ 1
Vax 785-Unix	***Franz-Lisp***	8 Mbytes	215	.64	182	none	
TI Explorer	***Common Lisp***	8 Mbytes	228	.60	228	none	
TI Explorer	***Zeta-Lisp***	8 Mbytes	230	.60	230	none	
Xerox 1186	***Interlisp***	3.5 Mbytes	348	.39	348	off	
Vax 780-Unix	***Franz-Lisp***	4 Mbytes	365	.38	298	??	~ 1
Xerox 1109	***Interlisp***	3.5 Mbytes	397	.35	397	off	
Xerox 1186	***Interlisp***	3.5 Mbytes	409	.33	366	43	
Xerox 1109	***Interlisp***	3.5 Mbytes	445	.31	402	43	
Vax 785-Unix	***Franz-Lisp***	8 Mbytes	470	.29	182	??	~ 3
Dec-2060	***Common Lisp***	8 Mbytes	660	.21	196	??	??
Vax 750-Unix	***Franz-Lisp***	4 Mbytes	676	.20	495	??	1

The fraction following the elapsed time is the elapsed time for the given machine divided by the elapsed time of the 3600. The performance of these systems may be different for other programs and even for other tasks in ***Soar*** that have different runtime characteristics than the Eight Puzzle. The Eight Puzzle task is CPU intensive, spending most of its time matching productions to working memory using a modified version of the ***Ops5*** Rete matcher. This uses simple symbolic computations, such as equality tests, function calls, application of functions (apply), and list manipulation. There is no number-crunching of integers or reals. A trace of the problem solving is printed to the terminal or console, but that is not a significant factor in any of the runs. There is no file input or output and all of the systems had enough memory so there was no within-process swapping.

All of the single-user workstations had sufficient virtual memory so that garbage collection was unnecessary. This is one of the biggest weaknesses of this benchmark because different types of garbage collectors are used by the different systems, with different overheads. For very long runs, garbage collection can become an important factor in performance. The Xerox machines have reference garbage collectors while the 3600 has an ephemeral garbage collector, both which are used incrementally (they do not wait for memory to get low before they run), so runs with their garbage collectors enabled were included. The elapsed time for the Xerox machines with their garbage collectors disabled is less than their CPU times using garbage collection because the CPU time includes some of the overhead associated with garbage collection (such as updating reference counts).

14. Soar Bibliography

Overview

Laird, J. E., Newell, A., & Rosenbloom, P. S. ***Soar:*** An Architecture for General Intelligence. 1986. In preparation.

This is a comprehensive scientific description of ***Soar*** (***Soar 4***) and the major research results.

Laird, J. E., Newell, A., & Rosenbloom, P. S. Proposal for Research on ***Soar***: An Architecture for General Intelligence and Learning. 1985.

This proposal provides a description of the research approach, a review of the principal research results, a survey of related research, and proposed research for the period 1985-1988.

Major Components

Problem Spaces

Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII*. Hillsdale, N.J.: Erlbaum, 1980. (Also available as CMU CSD Technical Report, Aug 79).

This paper lays out the foundations behind the use of problem spaces for all goal-oriented behavior.

Universal Weak Method

Laird, J. E., and Newell, A. *A Universal Weak Method* (Tech. Rep. #83-141). Carnegie-Mellon University Computer Science Department, June 1983.

Discusses the weak methods, the problem-space hypothesis, ***Soar1***, what a universal weak method is, a particular universal weak method, and a demonstration of it involving the use of many methods on many tasks in ***Soar1***. (***Soar1*** differs significantly from the version of ***Soar*** described in this manual.)

Laird, J. E., and Newell, A. A universal weak method: Summary of results. In *Proceedings of the Eighth IJCAI*. 1983.

A summary of the longer universal weak method paper.

Universal Subgoaling

Laird, J. E. *Universal Subgoaling*. Doctoral dissertation. Carnegie-Mellon University, 1983. (Available as Carnegie-Mellon University Computer Science Tech. Rep. #84-129).

Discusses the concept of universal subgoaling, updates the universal weak method to use universal subgoaling, presents ***Soar2*** and some demonstrations of it.(***Soar2*** differs significantly from the version of ***Soar*** described in this manual.)

Chunking

Rosenbloom, P. S., and Newell, A. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach. Volume II*. Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1986.

This paper lays out the foundations for goal-based chunking (in the context of the ***Xaps3*** architecture).

Laird, J. E., Rosenbloom, P. S., & Newell, A. Towards chunking as a general learning mechanism. In

Proceedings of AAAI-84, National Conference on Artificial Intelligence. American Association for Artificial Intelligence, 1984. Available in *Two **Soar** Studies*, (Tech. Rep. #85-110). Carnegie-Mellon University Computer Science Department, January 1985.

This paper presents the first results from implementing chunking in ***Soar***: strategy acquisition, normal practice speed-ups, within-trial transfer, across-task transfer, and knowledge acquisition.

Rosenbloom, P. S., Laird, J. E., Newell, A., Golding, A., Unruh, A. Current research on learning in Soar. In *Proceedings of the Third International Machine Learning Workshop*, 1985, Skytop, PA.

This paper reviews the state of research on chunking in Soar as of July, 1985. It includes short discussions of work on analogy and generalization, simple abstraction planning, macro-operator acquisition, and problem space creation.

Laird, J. E., Rosenbloom, P. S., & Newell, A. Chunking in Soar: The anatomy of a general learning mechanism. In *Machine Learning*, 1986 1(1) 11-44.

This paper presents the details of chunking in ***Soar***. It includes a demonstration of chunking based on Korf's Macro Problem Solver.

Manuals

Laird, J. E. ***Soar** User's Manual*. Version 4. 1986.

The manual is the main reference for using ***Soar 4***.

Laird, J. E. ***Soar** Technical Manual*. 1985. In preparation.

The manual is the main reference for the ***Soar*** software.

Forgy, C. L. ***Ops5** Manual*. Computer Science Department, Carnegie-Mellon University, 1981.

Soar is implemented on top of ***Ops5***, and thus inherits many aspects of it.

Applications

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orciuch, E. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1985 7(5) 561-569. This also appeared in *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*. IEEE Computer Society, 1984. Available in *Two **Soar** Studies*, (Tech. Rep. #85-110). Carnegie-Mellon University Computer Science Department, January 1985.

This paper presents the first attempt at expert systems in ***Soar***, a partial reimplementation of ***R1***. It shows how problem solving and expertise can be integrated, and how chunking can acquire expertise from problem solving.

Appendix I
Default Search-Control Productions

Below are the default productions in default.soar.

```
(comment ****** common search-control productions ******)

(comment all operator augmentations of the problem space have
         acceptable-preferences created for them)

(sp default*make-all-operators-acceptable
  (gc <g> ↑problem-space <p>)
  (problem-space <p> ↑operator <x>)
 -(preference <x> ↑role operator ↑value acceptable ↑problem-space <p>)
  -->
  (preference <x> ↑role operator ↑value acceptable
    ↑problem-space <p>))


(comment if an operator has just been applied to a state, which is
         detected by using the preference created for that state,
         reject the operator for that state so it will not be reapplied
         in the future)

(sp default*no-operator-retry
    (gc <g> ↑problem-space <p> ↑state <s2>)
    (preference ↑object <s2> ↑role state ↑value acceptable
                ↑goal <g> ↑problem-space <p> ↑state <s>
                ↑operator { <> undecided <> nil <o> })
    -->
    (preference <o> ↑role operator ↑value reject
          ↑goal <g> ↑problem-space <p> ↑state <s>))


(comment if there is a reject-preference for the current state,
         make an acceptable-preference for the prior state so problem
         solving can backup)

(sp default*backup-if-failed-state
    (gc <g> ↑problem-space <p> ↑state <s>)
    (preference <s> ↑role state ↑value reject
                ↑goal <g> ↑problem-space <p>)
    (preference <s> ↑role state ↑value acceptable
                ↑goal <g> ↑problem-space <p> ↑state { <> undecided <> nil <n> }
                ↑operator <> undecided)
    -->
    (preference <n> ↑role state ↑value acceptable
          ↑goal <g> ↑problem-space <p> ↑state <s>))
```

```
(comment ****** default knowledge for tie impasses ******)

(comment if the problem space for handling the subgoal fails,
         signified by the choices none impasse below it,
         make a worst-preference for each tied object)

 (sp default*problem-space-tie
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role problem-space ^impasse tie ^supergoal <g1>
             ^item <p>)
     -->
    (preference <p> ^role problem-space ^value worst
             ^goal <g1>))

(sp default*state-tie
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role state ^impasse tie ^supergoal <g1> ^item <s>)
    (gc <g1> ^problem-space <p>)
     -->
    (preference <s> ^role state ^value worst
             ^goal <g1>))

(sp default*operator-tie
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role operator ^impasse tie ^supergoal <g1> ^item <o>)
    (gc <g1> ^problem-space <p> ^state <s>)
     -->
    (preference <o> ^role problem-space ^value worst
             ^goal <g1> ^problem-space <p>))


(comment ****** conflict impasses ******)

(comment if the problem space for handling the subgoal fails,
         signified by the choices none impasse below it,
         make a reject-preference for each conflicted object)

 (sp default*problem-space-conflict
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role problem-space ^impasse conflict ^supergoal <g1>
             ^item <p>)
     -->
    (preference <p> ^role problem-space ^value reject
             ^goal <g1>))

(sp default*state-conflict
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role state ^impasse conflict
             ^supergoal <g1> ^item <s>)
    (gc <g1> ^problem-space <p>)
     -->
    (preference <s> ^role state ^value reject
             ^goal <g1> ^problem-space <p>))

(sp default*operator-conflict
    (gc <g3> ^role goal ^choices none ^supergoal <g2>)
    (gc <g2> ^role operator ^impasse conflict ^supergoal <g1>
             ^item <o>)
    (gc <g1> ^problem-space <p> ^state <s>)
     -->
    (preference <o> ^role operator ^value reject
             ^goal <g1> ^problem-space <p> ^state <s>))
```

```
(comment ****** no-choice impasses ******)

(comment if no problem spaces are available for the top goal,
         terminate the problem solving session with halt)

(sp default*goal-no-choices
    (gc <g3> ↑role goal ↑choices none ↑supergoal <g2>)
   -(gc <g2> ↑supergoal)
    -->
    (write1 (crlf) "no problem space can be selected for top goal.")
    (write1 (crlf) "soar must terminate.")
    (halt))


(comment if no states are available for a problem space,
         and there is no problem space to find more,
         reject that problem space)

(sp default*problem-space-no-choices
    (gc <g3> ↑role goal ↑choices none ↑supergoal <g2>)
    (gc <g2> ↑role problem-space ↑choices none ↑supergoal <g1>)
    (gc <g1> ↑problem-space <p>)
    -->
    (preference <p> ↑role problem-space ↑value reject ↑goal <g1>))


(comment if no operators are available for a state,
         and there is no problem space to find more,
         reject that state)

(sp default*state-no-choices
    (gc <g3> ↑role goal ↑choices none ↑supergoal <g2>)
    (gc <g2> ↑role state ↑choices none ↑supergoal <g1>)
    (gc <g1> ↑problem-space <p> ↑state <s>)
    -->
    (preference <s> ↑role state ↑value reject
                ↑goal <g1> ↑problem-space <p>))

(comment if no changes for an operator,
         and there is no problem space to find more,
         reject that operator)

(sp default*operator-no-choices
    (gc <g3> ↑role goal ↑choices none ↑supergoal <g2>)
    (gc <g2> ↑role operator ↑impasse no-change ↑supergoal <g1>)
    (gc <g1> ↑problem-space <p> ↑state <s> ↑operator <o>)
    -->
    (preference <o> ↑role operator ↑value reject
          ↑goal <g1> ↑problem-space <p> ↑state <s>))
```

```
(comment ************** selection problem space ****************)


(comment use the selection problem space for all choice multiple
         impasses.  make it worst so that any other will dominate)

(sp select*selection-space elaborate
    (gc <g> ^choices multiple)
    -->
    (preference <p> ^role problem-space ^value acceptable ^goal <g>)
    (preference <p> ^role problem-space ^value worst ^goal <g>)
    (problem-space <p> ^name selection))


(comment the state of the selection problem space is empty)

(sp select*create-state
   (gc <g> ^problem-space <p> ^state undecided)
   (space <p> ^name selection)
   -->
   (preference <s> ^role state ^value acceptable
        ^goal <g> ^problem-space <p> ^state undecided))



(comment ************** evaluate-object operator ****************))


(comment create an evaluate-object operator for each tying item
         in selection problem space.  These are all indifferent
         so there will be no tie between them.)

(sp eval*select-evaluate
   (gc <g> ^problem-space <p> ^state <s> ^supergoal <g2> ^item <x>)
   (problem-space <p> ^name selection)
   -->
   (operator <o> ^state <s> ^name evaluate-object ^object <x>)
   (preference <o> ^role operator ^value indifferent
        ^goal <g> ^problem-space <p> ^state <s> )
   (preference <o> ^role operator ^value acceptable
        ^goal <g> ^problem-space <p> ^state <s> ))


(comment for parallel evaluation
         remove this comment if you want parallel evaluation of
         the alternatives.

(sp eval*parallel-evaluate
   (gc <g> ^problem-space <p> ^state <s> ^role operator ^supergoal <g2>)
   (problem-space <p> ^name selection)
   (preference <o1> ^role operator ^value acceptable
        ^goal <g> ^problem-space <p> ^state <s>)
   (preference <o2> ^role operator ^value acceptable
        ^goal <g> ^problem-space <p> ^state <s>)
   (operator <o1> ^object <y>)
   (operator <o2> ^object { <> <y> <x> })
   -->
   (preference <o1> ^role operator ^value parallel
        ^goal <g> ^problem-space <p> ^state <s> ^reference <o2>)))
```

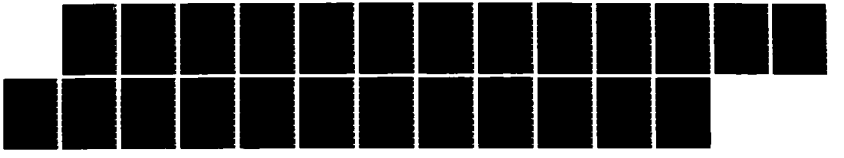
```
(comment create evaluation once the eval operator is selected)

(sp eval*apply-evaluate
   (gc <g> tproblem-space <p> tstate <s> toperator <o>
        trole <role> tsupergoal <g2>)
   (problem-space <p> tname selection)
   (gc <g2> tproblem-space <p2> tstate <s2> tdesired <d>)
   (operator <o> tname evaluate-object tobject <x>)
   -->
   (state <s> tevaluation <e>)
   (evaluation <e> tobject <x> tstate <s> toperator <o> tdesired <d>)
   (operator <o> trole <role> tevaluation <e> tdesired <d>
        tsupergoal <g2> tsuperproblem-space <p2> tsuperstate <s2>))


(comment reject evaluate-object after it finished in selection space)

(sp eval*reject-evaluate-finished
   (gc <g> tproblem-space <p> tstate <s> toperator <o>)
   (problem-space <p> tname selection)
   (operator <o> tname evaluate-object tevaluation <e>)
   (evaluation <e> t << numeric-value symbolic-value >>)
   -->
   (preference <o> trole operator tvalue reject tgoal <g>
         tproblem-space <p> tstate <s>))
```

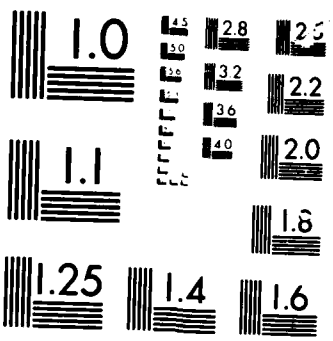
```
(comment if two objects have equal evaluations they are indifferent)

(sp eval*equal-eval-indifferent-preference
   (gc <g> ↑problem-space <p> ↑state <s> ↑role <role> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (state <s> ↑evaluation <e1> ↑evaluation { <> <e1> <e2> })
   (gc <g2> ↑problem-space <p2> ↑state <s2> ↑desired <d>)
   (evaluation <e1> ↑object <x> ↑numeric-value <v> ↑desired <d>)
   (evaluation <e2> ↑object <y> ↑numeric-value <v> ↑desired <d>)
   -->
   (preference <x> ↑role <role> ↑value indifferent ↑reference <y>
         ↑goal <g2> ↑problem-space <p2> ↑state <s2>))


(comment generate operator preferences based on their evaluations and info
         as to whether higher or lower evaluations are better.)

(sp eval*prefer-higher-evaluation
   (gc <g> ↑problem-space <p> ↑state <s> ↑role <role> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (gc <g2> ↑problem-space <p2> ↑state <s2> ↑desired <d>)
   (state <s> ↑evaluation <e1> ↑evaluation { <> <e1> <e2> })
   (evaluation <d> ↑better higher)
   (evaluation <e1> ↑object <o1> ↑numeric-value <v> ↑desired <d>)
   (evaluation <e2> ↑object <o2> ↑numeric-value < <v> ↑desired <d>)
   -->
   (preference <o2> ↑role <role> ↑value worse ↑reference <o1>
         ↑goal <g2> ↑problem-space <p2> ↑state <s2>))

(sp eval*prefer-lower-evaluation
   (gc <g> ↑problem-space <p> ↑state <s> ↑role <role> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (gc <g2> ↑problem-space <p2> ↑state <s2> ↑desired <d>)
   (state <s> ↑evaluation <e1> ↑evaluation { <> <e1> <e2> })
   (evaluation <d> ↑better lower)
   (evaluation <e1> ↑object <o1> ↑numeric-value <v> ↑desired <d>)
   (evaluation <e2> ↑object <o2> ↑numeric-value > <v> ↑desired <d>)
   -->
   (preference <o2> ↑role operator ↑value worse ↑reference <o1>
         ↑goal <g2> ↑problem-space <p2> ↑state <s2>))
```

```
(comment ****** productions for the evaluation subgoal ******)

(comment copy down the desired and create the appropriate context,
         given the role of the object being evaluated)

(sp eval*select-role-problem-space
   (gc <g> tproblem-space undecided tsupergoal <g2> tsuperoperator <o2>)
   (gc <g2> toperator <o2>)
   (operator <o2> tname evaluate-object trole problem-space tobject <p> tdesired <d>)
   -->
   (gc <g> tdesired <d>)
   (preference <p> trole problem-space tvalue acceptable tgoal <g>))


(sp eval*select-role-state
   (gc <g> tproblem-space undecided tsupergoal <g2> tsuperoperator <o2>)
   (gc <g2> toperator <o2>)
   (operator <o2> tname evaluate-object trole state tobject <s>
        tsuperproblem-space <p> tdesired <d>)
   -->
   (gc <g> tdesired <d>)
   (preference <p> trole problem-space tvalue acceptable tgoal <g>)
   (preference <s> trole state tvalue acceptable
        tgoal <g> tproblem-space <p> tstate undecided)
   (preference <s> trole state tvalue best
        tgoal <g> tproblem-space <p> tstate undecided))


(sp eval*select-role-operator
   (gc <g> tproblem-space undecided tsupergoal <g2> tsuperoperator <o2>)
   (gc <g2> toperator <o2>)
   (operator <o2> tname evaluate-object trole operator tobject <o>
        tsuperproblem-space <p> tsuperstate <s> tdesired <d>)
   -->
   (gc <g> tdesired <d>)
   (preference <p> trole problem-space tvalue acceptable tgoal <g>)
   (preference <s> trole state tvalue acceptable
        tgoal <g> tproblem-space <p> tstate undecided)
   (preference <o> trole operator tvalue acceptable
         tgoal <g> tproblem-space <p> tstate <s>))


(comment reject those operators that are not being evaluated in this subgoal)

(sp eval*reject-non-slot-operator
   (gc <g> tproblem-space <p> tstate <s> tsupergoal <g2> tsuperoperator <o2>)
   (operator <o2> tname evaluate-object trole operator tobject <o>
         tsuperstate <s>)
   (preference { <> <o> <o3> } trole operator tvalue acceptable
         tgoal <g> tproblem-space <p> tstate <s>)
   -->
   (preference <o3> trole operator tvalue reject
         tgoal <g> tproblem-space <p> tstate <s>))
```

AD-A169 005 SOAR USER'S MANUAL(U) XEROX PALO ALTO RESEARCH CENTER CA INTELLIGENT SYSTEMS LAB J E LAIRD 31 JAN 86 ISL-15 N00014-82-C-0067 2/2

UNCLASSIFIED F/G 5/10 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0

```
(comment give symbol-value failure to an operator that has been rejected
         during evaluation and did not create a new state and reject the eval-operator)

(sp eval*failure-if-reject-evaling-operator
   (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o>
        ↑supergoal <g2> ↑superoperator <o2>)
   (gc <g2> ↑problem-space <p2> ↑state <s2>)
   (operator <o2> ↑name evaluate-object ↑role operator
        ↑object <o> ↑superstate <s> ↑evaluation <e2>)
   (preference <o> ↑role operator ↑value reject
        ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
  -(preference ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state <s> ↑operator <o>)
   -->
   (evaluation <e2> ↑symbolic-value failure))

(comment give symbol-value failure to an operator
         that produces a state that gets rejected in the subgoal)

(sp eval*failure-if-reject-state
   (gc <g> ↑problem-space <p> ↑state <s>
        ↑supergoal <g2> ↑superoperator <o2>)
   (gc <g2> ↑problem-space <p2> ↑state <s2>)
   (operator <o2> ↑name evaluate-object ↑evaluation <e2>)
   (preference <s> ↑role state ↑value reject
        ↑goal <g> ↑problem-space <p>)
   -->
   (evaluation <e2> ↑symbolic-value failure))


(comment if an operator leads to success and it is being
         tried out in a subgoal to evaluate another operator,
         give that second operator a success evaluation also)

(sp eval*pass-back-success
   (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (operator <o> ↑name evaluate-object ↑evaluation <e1> ↑desired <eb>)
   (evaluation <e1> ↑symbolic-value success)
   (gc <g2> ↑superoperator <o3>)
   (operator <o3> ↑name evaluate-object ↑evaluation <e2> ↑desired <eb>)
   -->
   (evaluation <e2> ↑symbolic-value success))
```

```
(comment if an operator is evaluated to be lose or failure for
         the same desired as the supergoal,
         create a worst-preference for it)

(sp eval*failure-becomes-worst
   (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (gc <g2> ↑problem-space <p2> ↑state <s2> ↑desired <d>)
   (operator <o> ↑name evaluate-object ↑evaluation <e1> ↑desired <d>
        ↑role <role> ↑object <o1>)
   (evaluation <e1> ↑symbolic-value << lose failure >>)
   -->
   (preference <o1> ↑role operator ↑value worst
         ↑goal <g2> ↑problem-space <p2> ↑state <s2>))


(comment if an operator is evaluated to be success for
         the same desired as the supergoal,
         create a best-preference for it)

(sp eval*success-becomes-best
   (gc <g> ↑problem-space <p> ↑state <s> ↑operator <o> ↑supergoal <g2>)
   (problem-space <p> ↑name selection)
   (gc <g2> ↑problem-space <p2> ↑state <s2> ↑desired <d>)
   (operator <o> ↑name evaluate-object ↑evaluation <e1>
         ↑desired <d> ↑object <o1> ↑role <role>)
   (evaluation <e1> ↑symbolic-value success)
   -->
   (preference <o1> ↑role <role> ↑value best
         ↑goal <g2> ↑problem-space <p2> ↑state <s2>))
```

```
(comment convert state augmentations into evaluations)

(sp eval*state-to-symbolic-evaluation
   (gc <g> ↑problem-space <p> ↑state <s> ↑superoperator <so>)
   (operator <so> ↑name evaluate-object
        ↑evaluation <e> ↑desired <eb>)
   (state <s> ↑{ << success failure win draw lose } <svalue> } <eb> )
   -->
   (evaluation <e> ↑symbolic-value <svalue>))

(comment handle state augmentations dealing with goal
         termination for the top-level goal)

(sp eval*detect-success
  (gc <g> ↑state <s> ↑name <name> ↑desired <eb> -↑supergoal)
  (state <s> ↑success <eb>)
  -->
  (write1 (crlf) "goal" <name> "achieved")
  (halt))

(sp eval*detect-win
  (gc <g> ↑state <s> ↑name <name> -↑supergoal ↑desired <eb>)
  (state <s> ↑win <eb>)
  -->
  (write1 (crlf) "game" <name> "won")
  (halt))

(sp eval*detect-failure
  (gc <g> ↑state <s> ↑name <name> -↑supergoal ↑desired <eb>)
  (state <s> ↑failure <eb>)
  -->
  (preference <s> ↑role state ↑value reject
        ↑goal <g> ↑problem-space <p>))

(sp eval*detect-lose
  (gc <g> ↑state <s> ↑name <name> -↑supergoal ↑desired <eb>)
  (state <s> ↑lose <eb>)
  -->
  (write1 (crlf) "game" <name> "lost")
  (halt))
```

```
(comment two player games - win side oside lose)

(sp eval*move-side-to-eval
   (gc <g> ↑state <s> ↑superoperator <so>)
   (state <s> ↑oside <side> ↑ << lose win >>)
   (operator <so> ↑name evaluate-object ↑evaluation <e>)
   -->
   (evaluation <e> ↑side <side>))

(sp eval*winning-values
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g1> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (gc <g1> ↑problem-space <p1> ↑state <s1>)
   (state <s1> ↑side <side>)
   (operator <o> ↑name evaluate-object ↑evaluation <e> ↑object <o1> ↑role <role>)
   (evaluation <e> ↑symbolic-value win ↑side <side>)
   -->
   (preference <o1> ↑role <role> ↑value best
         ↑goal <g1> ↑problem-space <p1> ↑state <s1>))

(sp eval*winning-values2
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g1> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (gc <g1> ↑problem-space <p1> ↑state <s1>)
   (state <s1> ↑oside <side>)
   (operator <o> ↑name evaluate-object ↑evaluation <e> ↑object <o1> ↑role <role>)
   (evaluation <e> ↑symbolic-value lose ↑side <side>)
   -->
   (preference <o1> ↑role <role> ↑value best
         ↑goal <g1> ↑problem-space <p1> ↑state <s1>))

(sp eval*draw-values
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g1> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (gc <g1> ↑problem-space <p1> ↑state <s1>)
   (operator <o> ↑name evaluate-object ↑evaluation <e> ↑object <o1> ↑role <role>)
   (evaluation <e> ↑symbolic-value draw)
   -->
   (preference <o1> ↑role <role> ↑value indifferent
         ↑goal <g1> ↑problem-space <p1> ↑state <s1>))
```

```
(sp eval*losing-values
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g1> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (gc <g1> ↑problem-space <p1> ↑state <s1>)
   (state <s1> ↑side <side>)
   (operator <o> ↑name evaluate-object ↑evaluation <e> ↑object <o1> ↑role <role>)
   (evaluation <e> ↑symbolic-value win ↑side <side>)
   -->
   (preference <o1> ↑role <role> ↑value worst
         ↑goal <g1> ↑problem-space <p1> ↑state <s1>))

(sp eval*losing-values2
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g1> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (gc <g1> ↑problem-space <p1> ↑state <s1>)
   (state <s1> ↑side <side>)
   (operator <o> ↑name evaluate-object ↑evaluation <e> ↑object <o1> ↑role <role>)
   (evaluation <e> ↑symbolic-value lose ↑side <side>)
   -->
   (preference <o1> ↑role <role> ↑value worst
         ↑goal <g1> ↑problem-space <p1> ↑state <s1>))

(sp eval*pass-back-win
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g2> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (operator <o> ↑name evaluate-object ↑evaluation <e1> ↑desired <eb>)
   (evaluation <e1> ↑symbolic-value win ↑side <side>)
   (gc <g2> ↑superoperator <o3>)
   (operator <o3> ↑name evaluate-object ↑evaluation <e2> ↑desired <eb>
        ↑superstate <s4>)
   (state <s4> ↑side <side>)
   -->
   (evaluation <e2> ↑symbolic-value win ↑side <side>))

(sp eval*pass-back-win2
   (gc <g> ↑problem-space <p> ↑state <s> ↑supergoal <g2> ↑operator <o>)
   (problem-space <p> ↑name selection)
   (operator <o> ↑name evaluate-object ↑evaluation <e1> ↑desired <eb>)
   (evaluation <e1> ↑symbolic-value lose ↑side <side>)
   (gc <g2> ↑superoperator <o3>)
   (operator <o3> ↑name evaluate-object ↑evaluation <e2> ↑desired <eb>
        ↑superstate <s4>)
   (state <s4> ↑side <side>)
   -->
   (evaluation <e2> ↑symbolic-value win ↑side <side>))
```

```
(comment ***************** operator subgoaling *****************
         there are two ways to do operator subgoal
         just pass down most recent operator, or pass down all of them
         this implementation passes down just the super operator as the
         desired - uncomment opsub*go-for-it2 if you want all supergoals
         to be included)

(comment make the super-problem space the default
         when there is a no-change for the operator)

(sp opsub*try-operator-subgoaling
  (gc <g> ↑impasse no-change ↑role operator
        ↑problem-space undecided ↑supergoal <g2>)
  (gc <g2> ↑problem-space <p2>)
  -->
  (preference <p2> ↑goal <g> ↑role problem-space ↑value acceptable)
  (preference <p2> ↑goal <g> ↑role problem-space ↑value worst))


(comment if the superproblem-space is selected as the
         current problem space then operator subgoaling
         is being used so select the superstate -
         the superoperator becomes the desired)

(sp opsub*go-for-it
  (gc <g> ↑problem-space <p> ↑state undecided
        ↑impasse no-change ↑role operator ↑supergoal <g2>)
  (gc <g2> ↑problem-space <p> ↑state <s> ↑operator <o>)
  -->
  (gc <g> ↑name operator-subgoal ↑desired <o>)
  (preference <s> ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state undecided))


(comment  pass down all super operator subgoals as well
(sp opsub*go-for-it2
  (gc <g> ↑problem-space <p> ↑state undecided
        ↑impasse no-change ↑role operator ↑supergoal <g2>)
  (gc <g2> ↑problem-space <p> ↑state <s> ↑desired <o>)
  -->
  (gc <g> ↑desired <o>)) )


(comment don't select the operator for the initial state that we are
         subgoaling on)

(sp opsub*reject-opsub*operator
  (gc <g> ↑name operator-subgoal ↑problem-space <p> ↑state <s> ↑desired <o>)
  (preference <s> ↑role state ↑value acceptable
        ↑goal <g> ↑problem-space <p> ↑state undecided)
  -->
  (preference <o> ↑role operator ↑value reject
        ↑goal <g> ↑problem-space <p> ↑state <s>))
```

```
(comment select superoperator for all new states)

(sp opsub*select-opsub*operator
  (gc <g1> ↑name operator-subgoal ↑problem-space <p> ↑state <s> ↑desired <o>)
  -->
  (preference <o> ↑role operator ↑value acceptable
        ↑goal <g1> ↑problem-space <p> ↑state <s>)
  (preference <o> ↑role operator ↑value best
        ↑goal <g1> ↑problem-space <p> ↑state <s>))


(comment if superoperator applied to a state then success
         we make a preference for the state it created)

(sp opsub*detect-direct-opsub-success
  (gc <g0> ↑problem-space <p> ↑state <s> ↑operator <o>
        ↑supergoal <g1> ↑name operator-subgoal)
  (gc <g1> ↑problem-space <p> ↑state <s2> ↑operator <o>)
  (preference <ns> ↑role state ↑value acceptable
        ↑goal <g0> ↑problem-space <p> ↑state <s> ↑operator <o>)
  -->
  (preference <ns> ↑role state ↑value acceptable
        ↑goal <g1> ↑problem-space <p> ↑state <s2> ↑operator <o>))

(comment if there is an evaluation subgoal within
         an operator subgoal and the operator being
         subgoaled on is applied - success)

(sp opsub*detect-indirect-opsub-success
  (gc <g1> ↑name operator-subgoal ↑supergoal <g2>)
  (gc <g2> ↑problem-space <p> ↑state <s2> ↑operator <o>)
  (gc <g0> ↑problem-space <p> ↑state <s> ↑operator <o>
        ↑desired <o> ↑superoperator <so>)
  (operator <so> ↑name evaluate-object)
  (preference <ns> ↑role state ↑value acceptable
        ↑goal <g0> ↑problem-space <p> ↑state <s> ↑operator <o>)
  -->
  (state <s> ↑success <o>))


(comment if the operator being subgoaled on is the current
         operator and a no-change subgoal is created for it
         then reject it in the subgoal)

(sp opsub*reject-double-op-sub
  (gc <g1> ↑name operator-subgoal ↑desired <o>)
  (gc { <> <g1> <g3> } ↑name operator-subgoal)
  (gc <g3> ↑supergoal <g4>)
  (gc <g4> ↑problem-space <p> ↑state <s> ↑operator <o>)
 -(gc ↑supergoal <g3>)
  -->
  (preference <o> ↑role operator ↑value reject
        ↑goal <g4> ↑problem-space <p> ↑state <s>))
```

Appendix II
Summary of Functions and Variables

chunk-all-paths	Controls multiple chunks from different paths: nil
chunk-classes	SP classes that must appear in a chunk for it to be built: (state)
chunk-free-problem-spaces	Names of problem space not to chunk: ()
chunks	Names of chunks built: ()
max-chunk-conditions	The maximum number of conditions allowed in a chunk: 200
max-elaborations	The maximum number of elaboration cycles before a decision: 100
max-recurse	*Depth of look ahead used by ordering scheme:* 2
sp-classes	Association list of SP and P classes: ((gc . goal-context-info) ...)
spo-default-depth	Default depth that spo prints: 1
subgoal-tabs	If T, Watch 0 trace will tab in subgoals: T
warning	If nil, warnings will not be printed: T
watch-free-problem-spaces	List of problem space names not to trace: ()
back-trace	Print out those conditions and productions that lead to the action: (back-trace O0034)
cs	Print the conflict set: (cs)
d	Run *N* decision cycles: (d 5)
decide-trace	Trace the decision procedure, t or **nil**: (decide-trace nil)
excise	Remove a production from production memory: (excise eight*create-state)
excise-chunks	Excise all chunks: (excise-chunks)
init-context	*Initialize the top context:* (init-context 'p1 's1 'o1)
init-soar	Clear out working memory: (init-soar)
last-chunk	Print out most recently built chunk in SP format: (last-chunk)
learn	Control chunking: (learn off always print)
list-chunks	Print out chunks in SP format: (list-chunk)
make	Add element to working memory: (make state-info ↑identifier s02 ...)
matches	Show all working-memory elements that match a production: (matches eight*create-state)
multi-attributes	Declare some attributes of some classes to be sets: (multi-attributes '((state binding 9)))
p	*Define a production:* (p eight*create-state (goal-context-info ↑identifier <g> ...) ...)
pbreak	Break after production fires or context change:(pbreak evaluate-object eight*create-state)
pi	Print the *Nth* partial instantiation of a production: (pi eight*create-state 3)
pgs	Print the goal-context stack: (pgs)
pm	Print production in P format: (pm eight*create-state)
po	Print all augmentations of object: (po G0033)
pop-goal	Terminate all goal and its subgoals: (pop-goal g0045)
ppwm	Prettyprint working-memory elements: (ppwm state-info)
print-stats	Print statistics from a run: (print-stats)
ptrace	Trace a production, object or working-memory element: (ptrace eight*create-state)
restart-soar	Clear out production memory and working memory: (restart-soar)
run	Run *N* productions cycles: (run 100)
smake	Add element in SP format to working memory: (smake state s02 ↑av 3)
smatches	*Display part of production that matches:* (smatches eight*create-state)
soarload	Load in productions, especially for D-machines: (soarload 'default.soar)
sp	Define a production in SP format: (sp eight*create-state (gc <g> ...) ...)
spm	Print production in SP format: (spm eight*create-state)
spo	Print all augmentations of objects in SP format to given depth: (spo G0003 2)
spop	Print all preferences of objects in SP format to given depth: (spop G0003 2)
spr	Print in SP format of whatever is given as an argument: (spr O0003)
sppwm	Prettyprint working-memory elements in SP format: (ppwm state-info)
sremove	Remove working-memory element with given time-tag: (sremove 33)
swm	SP print the object in the identifier field of the element with the time-tag: (swm 454)
trace-attributes	Will trace the attributes of the classes: (trace-attributes ((operator module)))
unpbreak	Remove a breakpoint, nil removes all breaks: (unpbreak selection)
untrace	Removes all traces set by ptrace: (untrace)
user-select	*Change how indifferent-preferences are handled, 'first, nil = random, T = user,* (3 selection 1)
watch	Control tracing, -1, 0, 5, 1, 1 5, 2 (higher = more): (watch 0)
wm	Print working-memory elements with given time-tags: (wm 434 455)

Index

1985/10/11

Xerox PARC/J.S. Brown

Dr. Phillip L. Ackerman
University of Minnesota
Department of Psychology
Minneapolis, MN 55455

Personnel Analysis Division,
AF/MPXA
5C360, The Pentagon
Washington, DC 20330

Air Force Human Resources Lab
AFHRL/MPD
Brooks AFB, TX 78235

AFOSR,
Life Sciences Directorate
Bolling Air Force Base
Washington, DC 20332

Dr. Robert Ahlers
Code N711
Human Factors Laboratory
NAVTRAEQUIPCEN
Orlando, FL 32813

Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152

Dr. James Anderson
Brown University
Center for Neural Science
Providence, RI 02912

Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Nancy S. Anderson
Department of Psychology
University of Maryland
College Park, MD 20742

Dr. Steve Andriole
Perceptronics, Inc.
21111 Erwin Street
Woodland Hills, CA 91367-3713

Technical Director, ARI
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

Dr. Patricia Baggett
University of Colorado
Department of Psychology
Box 345
Boulder, CO 80309

Dr. Gautam Biswas
Department of Computer Science
University of South Carolina
Columbia, SC 29208

Dr. John Black
Yale University
Box 11A, Yale Station
New Haven, CT 06520

Arthur S. Blaiwes
Code N711
Naval Training Equipment Center
Orlando, FL 32813

Dr. Jeff Bonar
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Gordon H. Bower
Department of Psychology
Stanford University
Stanford, CA 94306

Dr. Robert Breaux
Code N-095R
NAVTRAEQUIPCEN
Orlando, FL 32813

Dr. John S. Brown
XEROX Palo Alto Research
Center
3333 Coyote Road
Palo Alto, CA 94304

Dr. Bruce Buchanan
Computer Science Department
Stanford University
Stanford, CA 94305

1985/10/11

Xerox PARC/J.S. Brown

Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. Pat Carpenter
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Chair, Department of
Computer Science
College of Arts and Sciences
Catholic University of
Sciences
America
Washington, DC 20064

Dr. Fred Chang
Navy Personnel R&D Center
Code 51
San Diego, CA 92152

Dr. Eugene Charniak
Brown University
Computer Science Department
Providence, RI 02912

Dr. Michelene Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Mr. Raymond E. Christal
AFHRL/MOE
Brooks AFB, TX 78235

Dr. Yee-Yeen Chu
Perceptronics, Inc.
21111 Erwin Street
Woodland Hills, CA 91367-3713

Dr. William Clancey
Computer Science Department
Stanford University
Stanford, CA 94306

Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

Dr. Stanley Collyer
Office of Naval Technology
Code 222
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. Natalie Dehn
Department of Computer and
Information Science
University of Oregon
Eugene, OR 97403

Dr. R. K. Dismukes
Associate Director for Life

AFOSR
Bolling AFB
Washington, DC 20332

Defense Technical
Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC
(12 Copies)

ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, MD 20014

Dr. K. Anders Ericsson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Pat Federico
Code 511
NPRDC
San Diego, CA 92152

Dr. Jerome A. Feldman
University of Rochester
Computer Science Department
Rochester, NY 14627

Dr. Paul Feltovich
Southern Illinois University
School of Medicine
Medical Education Department
P.O. Box 3926
Springfield, IL 62708

1985/10/11

Xerox PARC/J.S. Brown

Mr. Wallace Feurzeig
Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

Dr. Craig I. Fields
ARPA
1400 Wilson Blvd.
Arlington, VA 22209

Dr. Gerhard Fischer
University of Colorado
Department of Computer Science
Boulder, CO 80309

Dr. Kenneth D. Forbus
University of Illinois
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801

Dr. Carl H. Frederiksen
McGill University
3700 McTavish Street
Montreal, Quebec H3A 1Y2
CANADA

Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

Dr. R. Edward Geiselman
Department of Psychology
University of California
Los Angeles, CA 90024.

Dr. Michael Genesereth
Stanford University
Computer Science Department
Stanford, CA 94305

Dr. Dedre Gentner
University of Illinois
Department of Psychology
603 E. Daniel St.
Champaign, IL 61820

Chair, Department of
Computer Science
George Mason University
Fairfax, VA 22030

Dr. Robert Glaser
Learning Research
& Development Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Dr. Joseph Goguen
Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

Dr. Sherrie Gott
AFHRL/MODJ
Brooks AFB, TX 78235

Dr. Richard H. Granger
Department of Computer Science
University of California, Irvine
Irvine, CA 92717

Dr. Wayne Gray
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Bert Green
Johns Hopkins University
Department of Psychology
Charles & 34th Street
Baltimore, MD 21218

Dr. James G. Greeno
University of California
Berkeley, CA 94720.

Chair, Department of
Computer and Information
Systems
The George Washington
University
Washington, DC 20052

Dr. Henry M. Halff
Halff Resources, Inc.
4918 33rd Road, North
Arlington, VA 22207

Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

1986/10/11

Xerox PARC/J.S. Brown

Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

Dr. Geoffrey Hinton
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Dr. Jim Hollan
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. John Holland
University of Michigan
2313 East Engineering
Ann Arbor, MI 48109

Dr. Keith Holyoak
University of Michigan
Human Performance Center
330 Packard Road
Ann Arbor, MI 48109

Dr. Earl Hunt
Department of Psychology
University of Washington
Seattle, WA 98105

Dr. Ed Hutchins
Intelligent Systems Group
Institute for
Cognitive Science (C-015)
UCSD
La Jolla, CA 92093

Dr. Dillon Inouye
WICAT Education Institute
Provo, UT 84057

Dr. Marcel Just
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Thomas Kehler
TEKNOWLEDGE
525 University Avenue
Palo Alto, CA 94301

Dr. Dennis Kibler
University of California
Department of Information
and Computer Science
Irvine, CA 92717

Dr. David Kieras
University of Michigan
Technical Communication
College of Engineering
1223 E. Engineering Building
Ann Arbor, MI 48109

Dr. Janet L. Kolodner
Georgia Institute of Technology
School of Information
& Computer Science
Atlanta, GA 30332

Dr. Kenneth Kotovsky
Department of Psychology
Community College of
Allegheny County
800 Allegheny Avenue
Pittsburgh, PA 15233

Dr. Benjamin Kuipers
Department of Mathematics
Tufts University
Medford, MA 02155

Dr. Pat Langley
University of California
Department of Information
and Computer Science
Irvine, CA 92717

Dr. Jill Larkin
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Dr. Robert Lawler
Information Sciences, FRL
GTE Laboratories, Inc.
40 Sylvan Road
Waltham, MA 02254

1985/10/11

Xerox PARC/J.S. Brown

Dr. Paul E. Lehner
PAR Technology Corp.
7926 Jones Branch Drive
Suite 170
McLean, VA 22102

Dr. Alan M. Lesgold
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

Dr. Clayton Lewis
University of Colorado
Department of Computer Science
Campus Box 430
Boulder, CO 80309

Science and Technology Division
Library of Congress
Washington, DC 20540

Dr. Don Lyon
P. O. Box 44
Higley, AZ 85236

Dr. Sandra P. Marshall
Dept. of Psychology
San Diego State University
San Diego, CA 92182

Dr. Manton M. Matthews
Department of Computer Science
University of South Carolina
Columbia, SC 29208

Dr. Kathleen McKeown
Columbia University
Department of Computer Science
New York, NY 10027

Dr. Al Meyrowitz
Office of Naval Research
Code 1133
800 N. Quincy
Arlington, VA 22217-5000

Dr. Ryszard S. Michalski
University of Illinois
Department of Computer Science
1304 West Springfield Avenue
Urbana, IL 61801

Prof. D. Michie
The Turing Institute
36 North Hanover Street
Glasgow G1 2AD, Scotland
UNITED KINGDOM

Dr. George A. Miller
Department of Psychology
Green Hall
Princeton University
Princeton, NJ 08540

Dr. William Montague
NPRDC Code 13
San Diego, CA 92152

Dr. Tom Moran
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Chair, Department of
Computer Science
Morgan State University
Baltimore, MD 21239

Dr. Allen Munro
Behavioral Technology
Laboratories - USC
1845 S. Elena Ave., 4th Floor
Redondo Beach, CA 90277

Chair, Department of
Computer Science
U.S. Naval Academy
Annapolis, MD 21402

Dr. David Navon
Institute for Cognitive Science
University of California
La Jolla, CA 92093

Dr. Allen Newell
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. T. Niblett
The Turing Institute
36 North Hanover Street
Glasgow G1 2AD, Scotland
UNITED KINGDOM

1985/10/11

Xerox PARC/J.S. Brown

Dr. Donald A. Norman
Institute for Cognitive Science
University of California
La Jolla, CA 92093

Library, NPRDC
Code P201L
San Diego, CA 92152

Commanding Officer,
Naval Research Laboratory
Code 2627
Washington, DC 20390

Dr. Stellan Ohlsson
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

Office of Naval Research,
Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142
800 N. Quincy St.
Arlington, VA 22217-5000

Office of Naval Research,
Code 1142PT
800 N. Quincy Street
Arlington, VA 22217-5000
(6 Copies)

Dr. Nancy Pennington
University of Chicago
Graduate School of Business
1101 E. 58th St.
Chicago, IL 60637

Department of Operations Research,
Naval Postgraduate School
Monterey, CA 93940

Department of Computer Science,
Naval Postgraduate School
Monterey, CA 93940

Dr. Martha Polson
Department of Psychology
Campus Box 346
University of Colorado
Boulder, CO 80309

Dr. Peter Polson
University of Colorado
Department of Psychology
Boulder, CO 80309

Dr. Steven E. Poltrock
MCC
9430 Research Blvd.
Echelon Bldg #1
Austin, TX 78759-6509

Dr. Harry E. Pople
University of Pittsburgh
Decision Systems Laboratory
1360 Scaife Hall
Pittsburgh, PA 15261

Dr. Joseph Psotka
ATTN: PERI-1C
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

Dr. Lynne Reder
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. James A. Reggia
University of Maryland
School of Medicine
Department of Neurology
22 South Greene Street
Baltimore, MD 21201

Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

Dr. Lauren Resnick
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1985/10/11

Xerox PARC/J.S. Brown

Dr. Mary S. Riley
Program in Cognitive Science
Center for Human Information Processing
University of California
La Jolla, CA 92093

Dr. William B. Rouse
Georgia Institute of Technology
School of Industrial & Systems Engineering
Atlanta, GA 30332

Dr. David Rumelhart
Center for Human Information Processing
Univ. of California
La Jolla, CA 92093

Dr. Roger Schank
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

Dr. Walter Schneider
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

Dr. Judah L. Schwartz
MIT
20C-120
Cambridge, MA 02139

Dr. Michael G. Shafto
ONR Code 1142PT
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. Sylvia A. S. Shafto
National Institute of Education
1200 19th Street
Mail Stop 1806
Washington, DC 20208

Dr. T. B. Sheridan
Dept. of Mechanical Engineering
MIT
Cambridge, MA 02139

Dr. Ted Shortliffe
Computer Science Department
Stanford University
Stanford, CA 94305

Dr. Randall Shumaker
Naval Research Laboratory
Code 7510
4555 Overlook Avenue, S.W.
Washington, DC 20375-5000

Dr. Herbert A. Simon
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

Dr. Derek Sleeman
Stanford University
School of Education
Stanford, CA 94305

Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

Dr. Elliot Soloway
Yale University
Computer Science Department
P.O. Box 2158
New Haven, CT 06520

James J. Staszewski
Research Associate
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

Dr. Paul J. Sticha
Senior Staff Scientist
Training Research Division
HumRRO
1100 S. Washington
Alexandria, VA 22314

1985/10/11

Xerox PARC/J.S. Brown

Dr. Patrick Suppes
Stanford University
Institute for Mathematical
Studies in the Social Sciences
Stanford, CA 94305

Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

Dr. Kikumi Tatsuoka
CERL
252 Engineering Research
Laboratory
Urbana, IL 61801

Dr. Perry W. Thorndyke
FMC Corporation
Central Engineering Labs
1185 Coleman Avenue, Box 580
Santa Clara, CA 95052

Dr. Douglas Towne
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

Chair, Department of
Computer Science
Towson State University
Towson, MD 21204

Chair, Department of
Computer Science
University of Maryland,
Baltimore County
Baltimore, MD 21228

Chair, Department of
Computer Science
University of Maryland,
College Park
College Park, MD 20742

Dr. Kurt Van Lehn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

Dr. Keith T. Wescourt
FMC Corporation
Central Engineering Labs
1185 Coleman Ave., Box 580
Santa Clara, CA 95052

Dr. Barbara White
Bolt Beranek & Newman, Inc.
10 Moulton Street
Cambridge, MA 02238

Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152

Dr. Masoud Yazdani
Dept. of Computer Science
University of Exeter
Exeter EX4 4QL
Devon, ENGLAND

Mr. Carl York
System Development Foundation
181 Lytton Avenue
Suite 210
Palo Alto, CA 94301

Dr. Michael J. Zyda
Naval Postgraduate School
Code 52CK
Monterey, CA 93943

END

DTIC

7-86